Database Processing
CS 451 / 551

Lecture 15:
MVCC Design Decisions and Database Durability

Suyash Gupta
Assistant Professor
Distopia Labs and ONRG
Dept. of Computer Science
(E) suyash@uoregon.edu
(W) gupta-suyash.github.io

UNIVERSITY OF OREGON

Presentations for Assignments from Tomorrow.

Final Exam: Dec 8, 2025 at 8-10am

Syllabus → Main focus on course not covered in Quizes, but you should know index and storage.

Last Class

- We discussed MVCC.

- In MVCC, the DBMS maintains multiple physical versions of each record in the database

- Next, we look at MVCC design decisions.

MVCC Design Decisions

- What do we need to consider while designing an MVCC scheme?

- Concurrency Control Protocol
- Version Storage
- Garbage Collection
- Deletes

Concurrency Control Protocol

- **Approach 1: Timestamp Ordering**
  - Assign transactions timestamps that determine serial order.

- **Approach 2: Optimistic Concurrency Control**
  - Three-phase protocol that we learnt in T/O lecture.
  - Use private workspace for new versions.

- **Approach 3: Two-Phase Locking**
  - Transactions acquire lock on physical version before they can read/write a logical tuple.

Version Storage

- **How to store versions?**

Version Storage

- **How to store versions?**
- The DBMS uses the record's pointer field to create a version chain (chain of versions).
- This allows the database to find the version that is visible to a particular transaction at runtime.
- Indexes always point to the **head** of the chain.
- Different storage schemes determine where/what to store for each version.

Version Storage

- **What are the possible designs ?**

Version Storage

- Three possible designs:

- **Append-Only Storage →**
  - New versions are appended to the same table.

- **Time-Travel Storage →**
  - Old versions are copied to a separate table.

- **Delta Storage →**
  - Only the original value of the modified column is copied into a separate space.

Append-Only Storage

- Say this is our original table.

Object	Value	Pointer
A_0	100	
B_0	220	

Append-Only Storage

- Now, we have an update to **variable A, so link from previous version**.
- On every update, append a new version of the record into an empty space in the table.

Object	Value	Pointer
A_0	100	
B_0	220	
A_1	150	

Append-Only Storage

- Now, we have another update to **variable A, so link from previous version**.

Object	Value	Pointer
A_0	100	
B_0	220	
A_1	150	
A_2	75	

Append-Only Storage

- Now, we have an update to **variable B, so link from previous version**.

Object	Value	Pointer
A_0	100	
B_0	220	
A_1	150	
A_2	75	
B_1	250	

Time-Travel Storage

- Say this is our original table.

Main Table

Object	Value	Pointer
A_0	100	
B_0	220	

Time-Travel Storage

- On every update, copy the current version to the time-travel table and update pointers.
- Say, a new version for A.

Main Table

Object	Value	Pointer
A_1	150	
B_0	220	

Time-Travel Table

Object	Value	Pointer
A_0	100	

Time-Travel Storage

- Say, another new version for A.
- Notice that we are overwriting the version in the main table.

Main Table

Object	Value	Pointer
A_2	175	
B_0	220	

Time-Travel Table

Object	Value	Pointer
A_1	150	
A_0	100	

Delta Storage

- Say this is our original table.

Main Table

Object	Attr 1	Attr 2	Pointer
A_0	100	AA	
B_0	220	BD	

Delta Storage

- Say, we **only update Attr 1 for record A**.
- On each update, copy only the attributes that were modified to the delta storage and overwrite the master version.

Main Table

Object	Attr 1	Attr 2	Pointer
A_0	150	AA	
B_0	220	BD	

Delta Table

Object	Delta	Pointer
A_0	Attr 1 → 100	

Delta Storage

- Say, we **have another update for Attr 1 of record A.**

Main Table

Object	Attr 1	Attr 2	Pointer
A_2	250	AA	
B_0	220	BD	

Delta Table

Object	Delta	Pointer
A_1	Attr 1 → 150	
A_0	Attr 1 → 100	

Garbage Collection

- **How to garbage collect old versions?**

Garbage Collection

- **How to garbage collect old versions?**

- The DBMS needs to carefully remove physical versions from the database over time.

- No active transaction in the DBMS should be able to see a version going to be garbage collected.
  - For example: A version was created by an aborted transaction should be garbage collected.

- Two additional design decisions:
  - How to look for expired versions?
  - How to decide when it is safe to reclaim memory?

Garbage Collection

- **Two approaches**

- **Tuple-level →**
  - Use a background thread to find old versions by examining tuples directly

- **Transaction-level →**
  - Each transaction keeps track of its old versions so the DBMS does not have to scan tuples.

Tuple-Level GC

- Makes use of a separate thread(s) also known as **background thread**.

- Background thread **periodically scans the table** and looks for reclaimable versions.

Tuple-Level GC

- Transaction 1 → T_{begin} = 12.
- Transaction 2 → T_{begin} = 25.

Object	Value	Begin-TS	End-TS
A_0	100	1	9
B_0	150	1	9
B_1	68	10	20

Tuple-Level GC

- Transaction 1 → T_{begin} = 12.
- Transaction 2 → T_{begin} = 25.

Object	Value	Begin-TS	End-TS
A_0	100	1	9
B_0	150	1	9
B_1	68	10	20

If a background thread scans this table, what can it conclude.

Tuple-Level GC

- Transaction 1 → T_{begin} = 12.
- Transaction 2 → T_{begin} = 25.

Object	Value	Begin-TS	End-TS
B_1	68	10	20

If a background thread scans this table, what can it conclude.
It can delete A_0 and B_0.

Transaction-Level GC

- Each transaction **keeps track of its read/write set**.
- On commit/abort, the transaction provides this information to the **background thread**.
- The background thread periodically determines when all versions created by a finished transaction are no longer visible.

Transaction-Level GC

Time

T1:

Begin
read(**A**)
write(**A**)
read(**B**)
write(**B**)
Commit

Object	Value	Begin-TS	End-TS
A_1	100	1	-
B_5	150	8	-

Say, this transaction T1 starts at time 10.

Transaction-Level GC

Time

T1:

Begin
read(**A**)
write(**A**)
read(**B**)
write(**B**)
Commit

Object	Value	Begin-TS	End-TS
A_1	100	1	10
B_5	150	8	-
A_1	100	10	-

Creates a new version of A (and subsequently B) for itself.

Transaction-Level GC

Time

T1:

Begin
read(**A**)
write(**A**)
read(**B**)
write(**B**)
Commit

Object	Value	Begin-TS	End-TS
A_1	100	1	10
B_5	150	8	-
A_1	200	10	-

Creates a new version of A (and subsequently B) for itself.

Transaction-Level GC

Time

T1:

Begin
read(**A**)
write(**A**)
read(**B**)
write(**B**)
Commit

Object	Value	Begin-TS	End-TS
A_1	100	1	10
B_5	150	8	10
A_2	200	10	-
B_6	150	10	-

Creates a new version of A (and subsequently B) for itself.

Transaction-Level GC

Time

T1:

Begin
read(**A**)
write(**A**)
read(**B**)
write(**B**)
Commit

Object	Value	Begin-TS	End-TS
A_1	100	1	10
B_5	150	8	10
A_2	200	10	-
B_6	99	10	-

Creates a new version of A (and subsequently B) for itself.

Transaction-Level GC

T1:

Begin
read(**A**)
write(**A**)
read(**B**)
write(**B**)
Commit

Time

Object	Value	Begin-TS	End-TS
A_1	100	1	10
B_5	150	8	10
A_2	200	10	-
B_6	99	10	-

At Commit, T1 can inform the background thread of discarding older versions (A_1 and B_5).

MVCC Deletes

- A record should be physically removed from the database only when all versions of the deleted record are not visible.
- Once a record is deleted, no newer version of that record can be created.
- We need a way to denote that the record has been logically deleted at some point in time.

MVCC Deletes

- Two mechanisms for deletes:
- **Deleted Flag**:
  - Maintain a flag that indicates a record has been deleted after the newest physical version.
  - Can either be in record header or a separate column.

- **Tombstone Tuple**:
  - Create an empty physical version to indicate that a record is deleted.

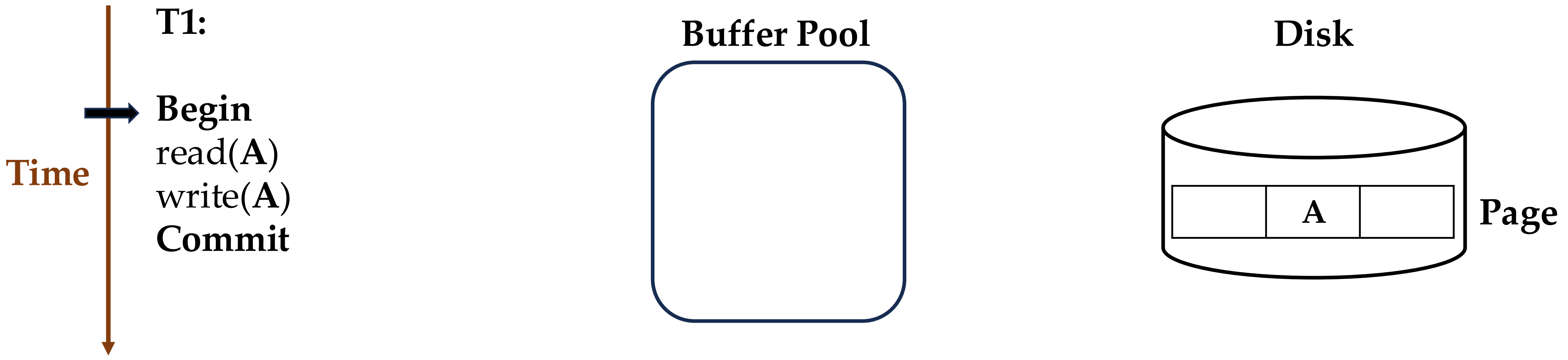
Durability

Durability

- Until now, we have discussed the **ACI of ACID properties**.

- DBMS needs to also provide Durability, that is, there is **no loss of data** despite any failures.
  - Failures should not cause loss of transaction commits.

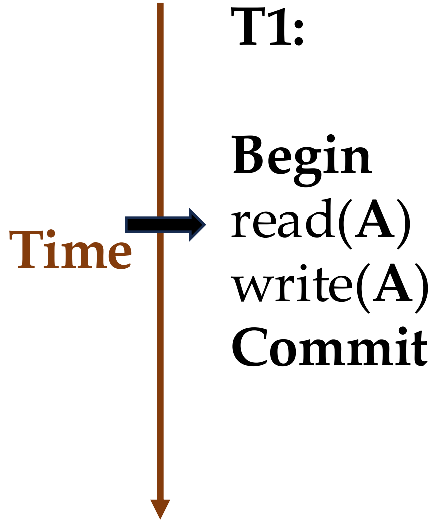
Lack of Durability

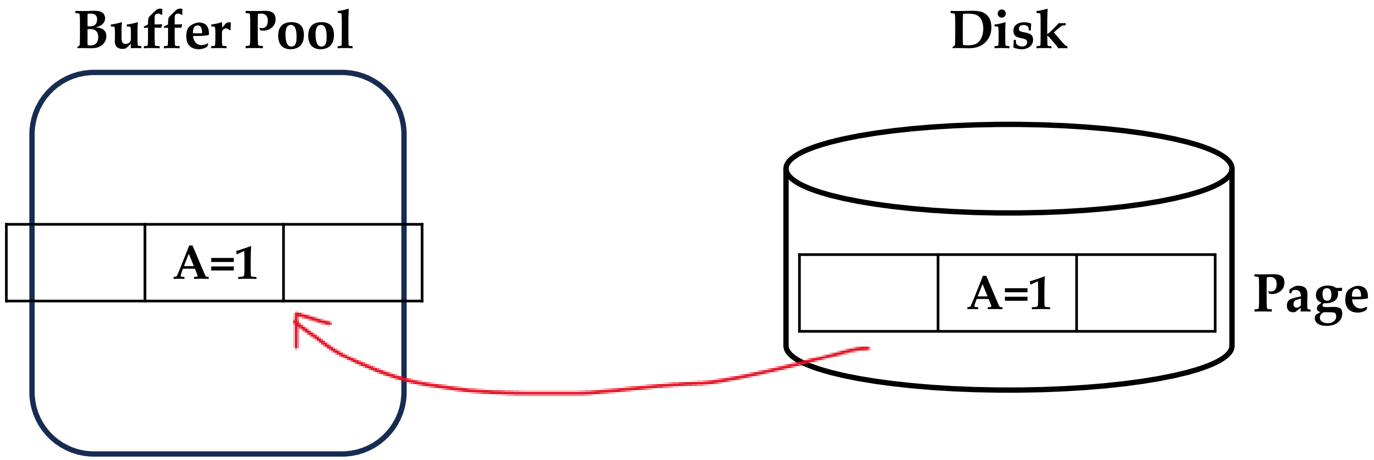


Let's try to understand the impact of lack of durability.

Recall that in every database we make use of a buffer pool for faster access.

Lack of Durability





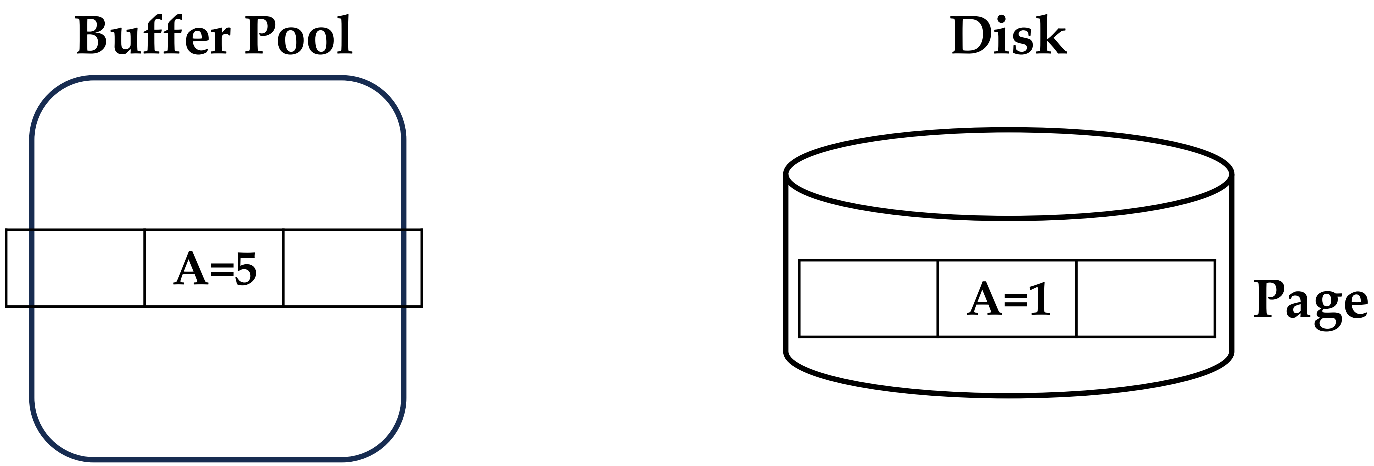
Read the page from disk to buffer pool.

Lack of Durability

Time

T1:

Begin
read(**A**)
write(**A**)
Commit



Update in the buffer pool.

Lack of Durability

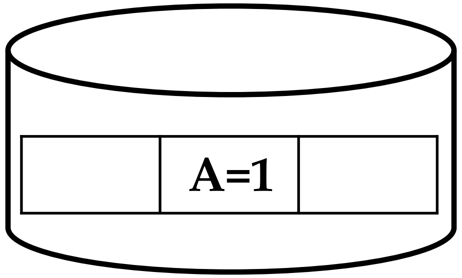
T1:

Begin
read(**A**)
write(**A**)
Commit

Time

Buffer Pool

Disk



Page

Say the system crashes → Memory wipes out → Buffer pool empty!

Durability

- Durability requirement necessitates **providing steps** that allows the database to recover from a failure.

- Remember, the primary storage location for any database is the nonvolatile storage (such as disk).

- However, disks are slower than volatile storage (main memory).

- So, steps:
  - First copy target record into memory.
  - Perform the writes in memory.
  - Write dirty records back to disk.

Undo vs. Redo

- **The DBMS needs to ensure that**
  - The changes for any transaction are durable once the DBMS announces that the transaction commits.
  - No partial changes are durable if the transaction aborts.

- **Undo:**
  - The process of removing the effects of an incomplete/ aborted transaction.

- **Redo:**
  - The process of re-applying the effects of a committed transaction.

Undo vs. Redo

- **The DBMS needs to ensure that**
  - The changes for any transaction are durable once the DBMS announces that the transaction commits.
  - No partial changes are durable if the transaction aborts.

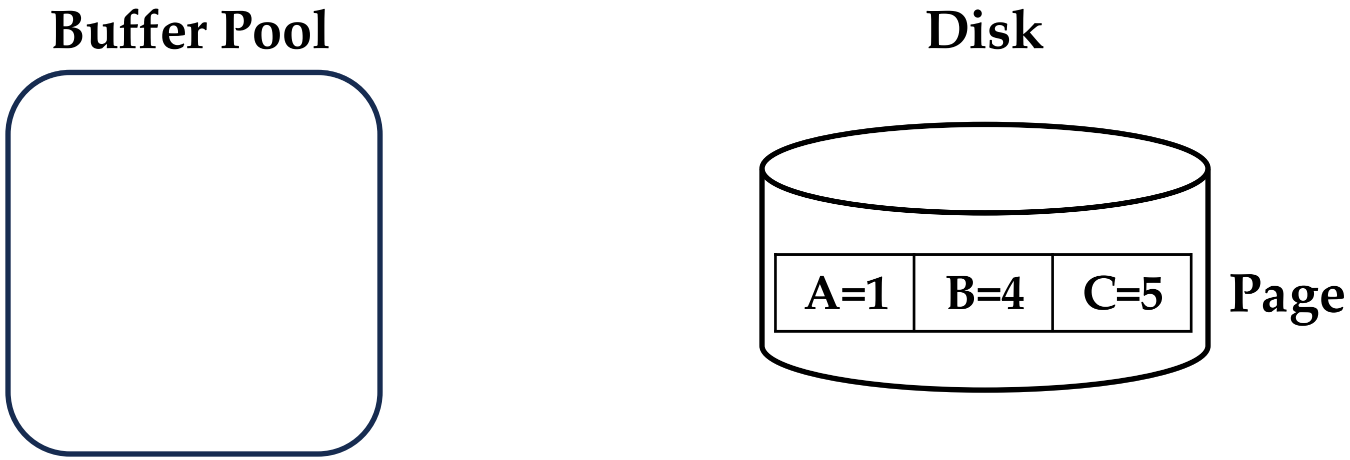
- **Undo:**
  - The process of removing the effects of an incomplete/ aborted transaction.

- **Redo:**
  - The process of re-applying the effects of a committed transaction.

- Whether to undo or redo depends on how the DBMS manages its buffer pool.

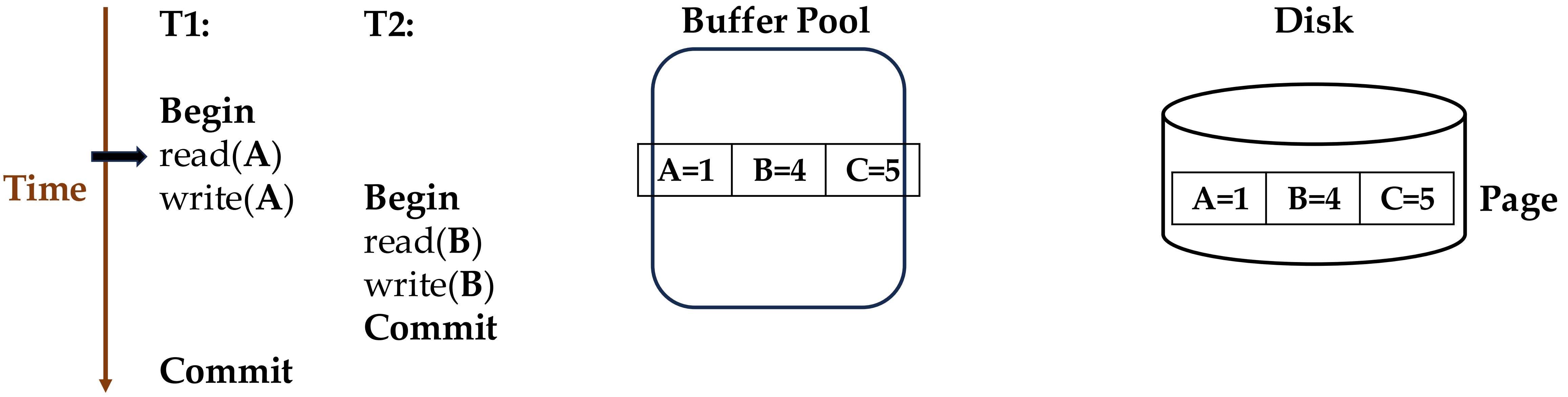
When to Write to Disk?

Time	T1:	T2:
	Begin	
	read(**A**)	
	write(**A**)	**Begin**
		read(**B**)
		write(**B**)
		Commit
	Commit	



Assume these are the two transactions running in our database.

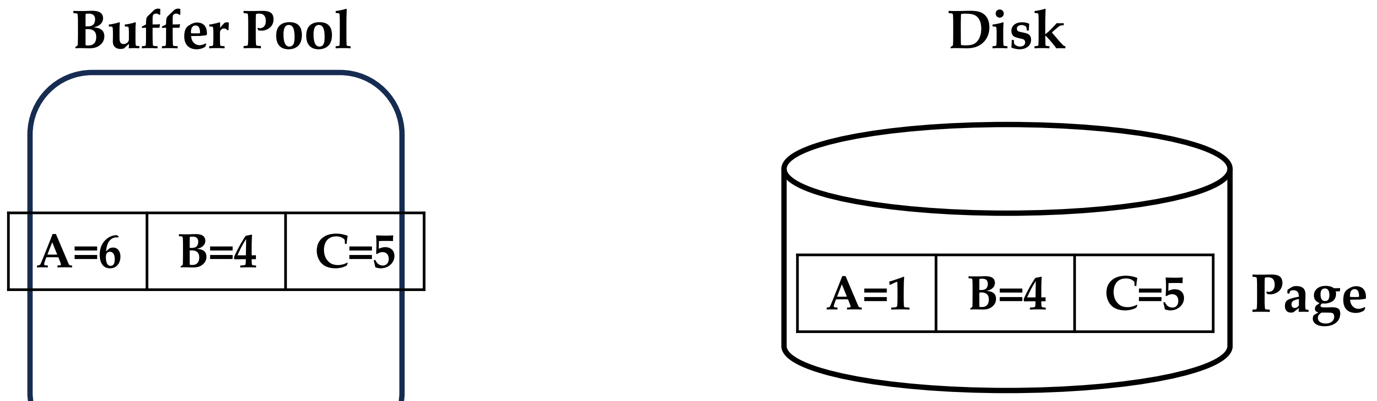
When to Write to Disk?



Read the page to the buffer.

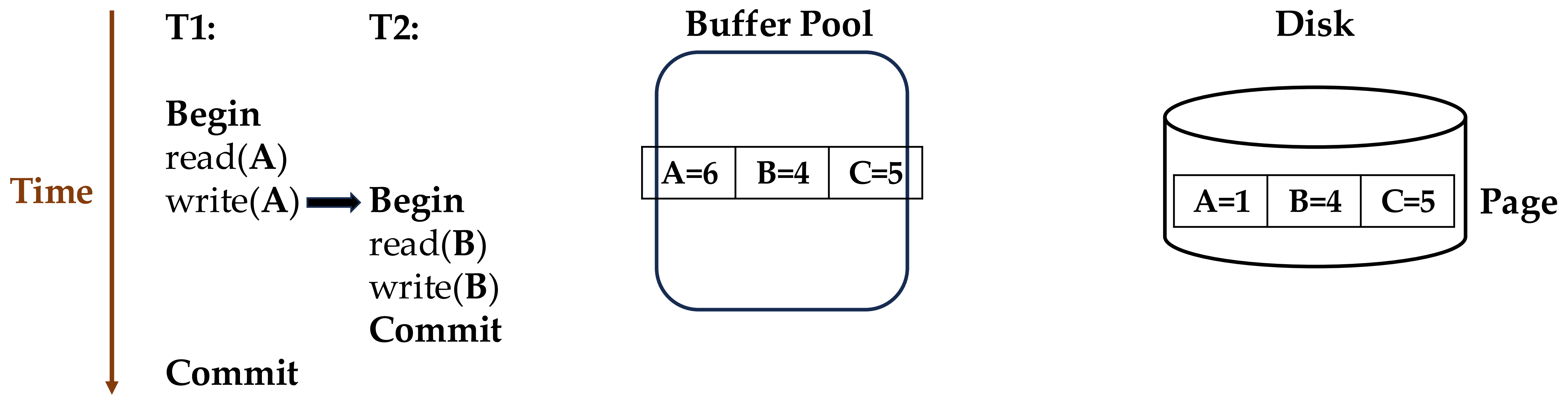
When to Write to Disk?

Time	T1:	T2:
	Begin	
	read(**A**)	
	write(**A**)	**Begin**
		read(**B**)
		write(**B**)
		Commit
	Commit	



Update A.

When to Write to Disk?

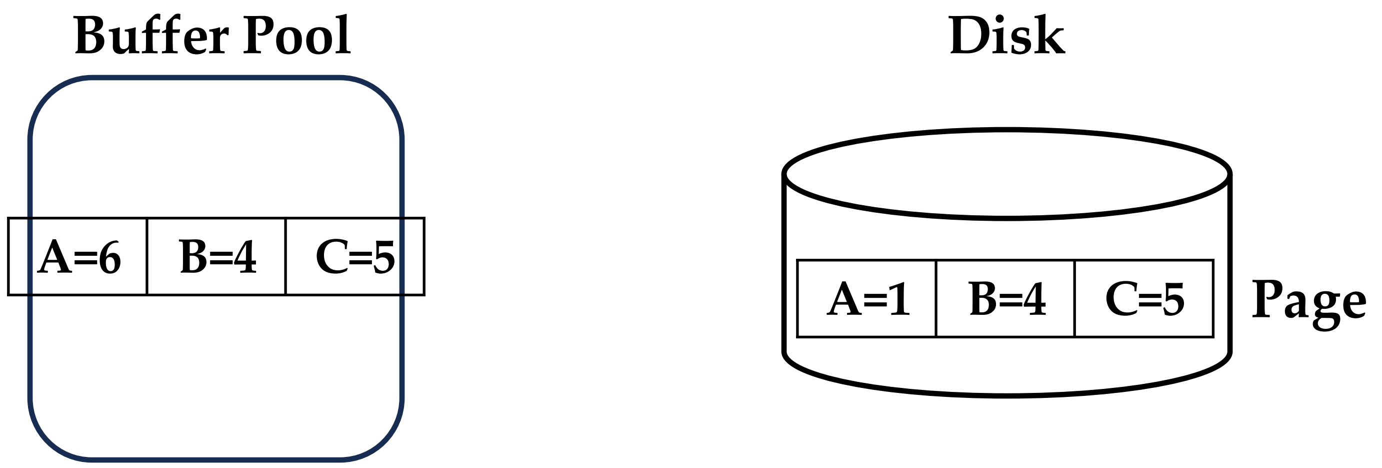


Records that T2 touches already in the buffer.

When to Write to Disk?

Time

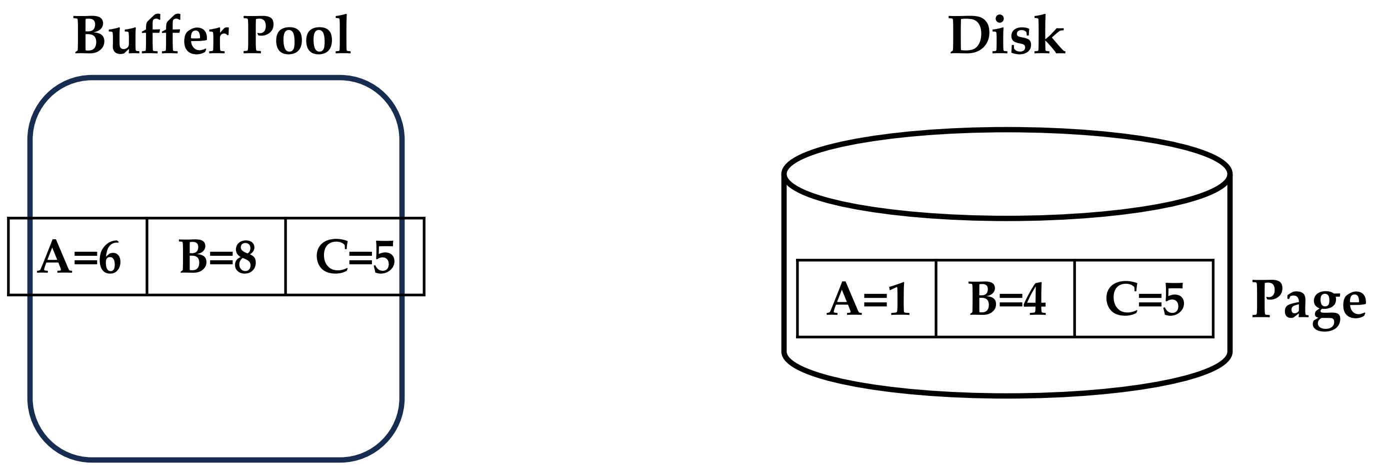
T1:	T2:
Begin	
read(**A**)	
write(**A**)	**Begin**
	read(**B**)
	write(**B**)
	Commit
Commit	



Records that T2 touches already in the buffer.

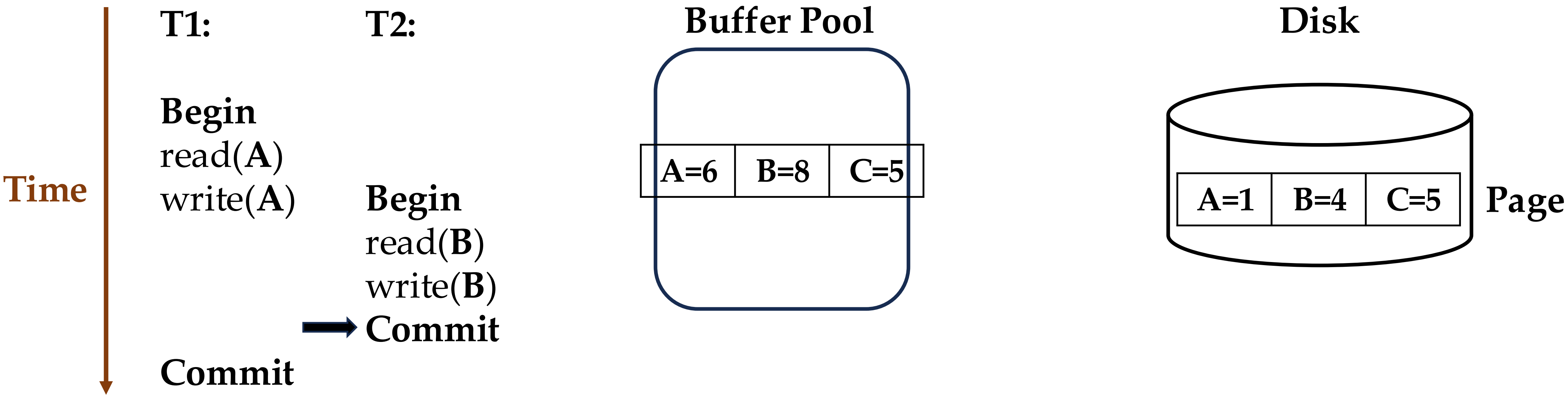
When to Write to Disk?

Time	T1:	T2:
	Begin	
	read(**A**)	
	write(**A**)	**Begin**
		read(**B**)
		write(**B**)
		Commit
	Commit	



Update B.

When to Write to Disk?



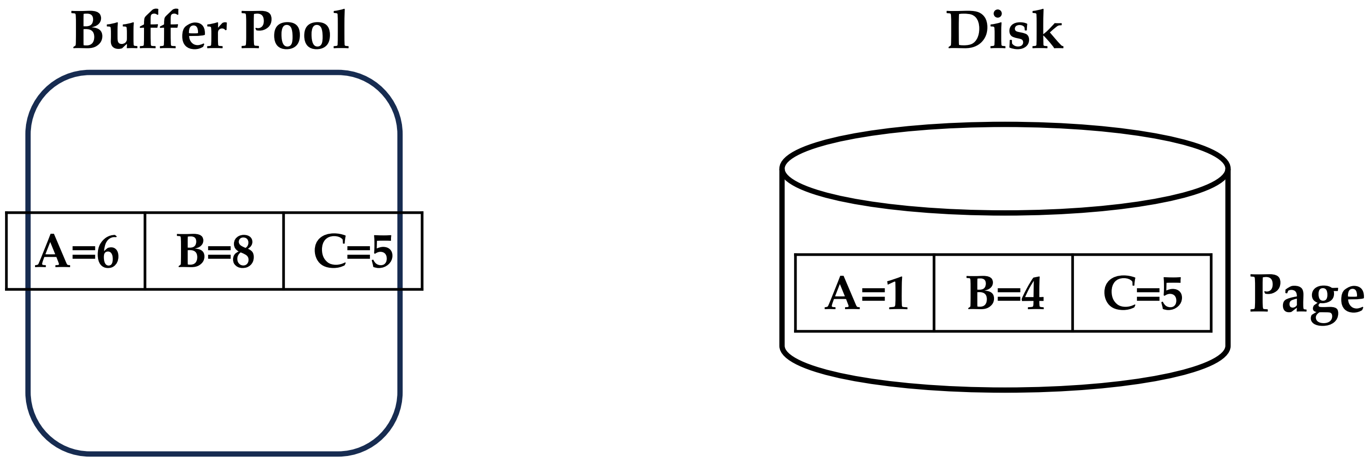
Time to commit B.

What should we do? How to make this transaction's updates durable?

When to Write to Disk?

Time

T1:	T2:
Begin	
read(**A**)	
write(**A**)	Begin
	read(**B**)
	write(**B**)
	Commit
Commit	

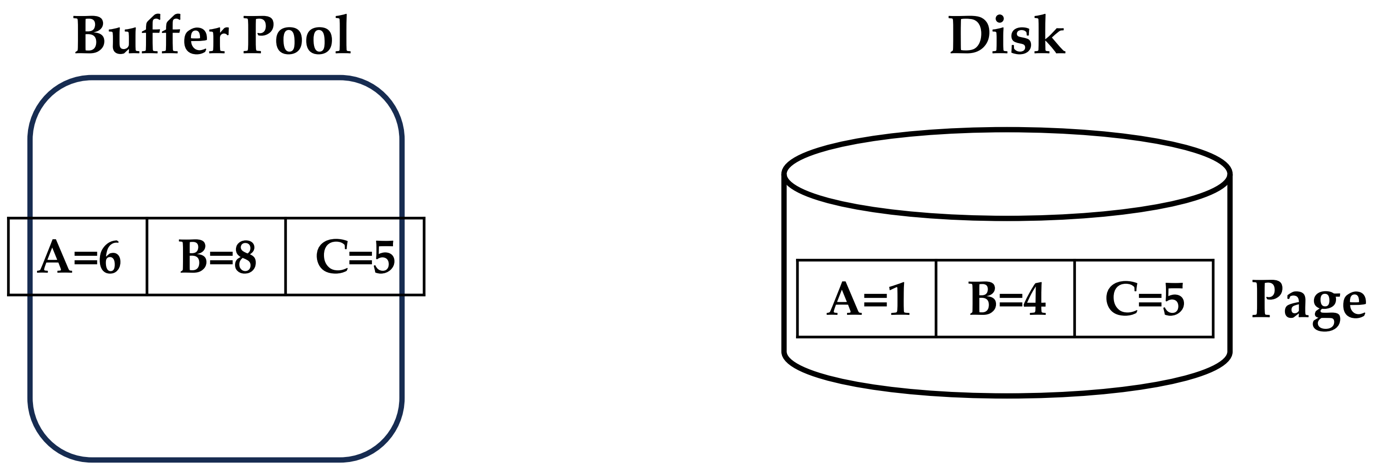


If we write this page from buffer back to disk, then we are writing changes of an uncommitted transaction (T1)!

When to Write to Disk?

Time

T1:	T2:
Begin	
read(**A**)	
write(**A**)	**Begin**
	read(**B**)
	write(**B**)
	Commit
Commit	



If we delay and the system crashes after commit of T2 but before commit of T1, then we lose durability!

Eviction Policies

- We need to design appropriate **Eviction Policies**.

- When/How to evict a page from the buffer pool.

- **Two design choices**:
  - Steal
  - Force

Steal Policy

- Steal Policy helps to determine:
  - Whether the DBMS can evict a dirty object in the buffer pool modified by an uncommitted transaction and use this dirty object to overwrite the most recent committed version of that object in the disk.

- **Steal:** Eviction + overwriting is allowed.
- **No-Steal:** No Eviction + No overwriting.

Force Policy

- Force policy helps to determine:
  - Whether the DBMS requires all updates made by a transaction are written back to the disk before the transaction can commit.

- **Force:** Write-back is required.
- **No-Force:** Write-back is not required.

Eviction Policies

Runtime Performance

	No-Steal	Steal
No-Force		Fastest
Force	Slowest	

Recovery Performance

	No-Steal	Steal
No-Force		Slowest
Force	Fastest	

No-Steal + Force

- No dirty write on an uncommitted transaction is written to the database and each update made by a transaction are written to the disk before commit.

No-Steal + Force

- No dirty write on an uncommitted transaction is written to the database and each update made by a transaction are written to the disk before commit.

- This approach is easy to implement:
  - Never have to undo changes of an aborted transaction because the changes were not written to disk.

  - Never have to redo changes of a committed transaction because all the changes are guaranteed to be written to disk at commit time.

- **How to design a no-steal + force policy**?

Shadow Paging

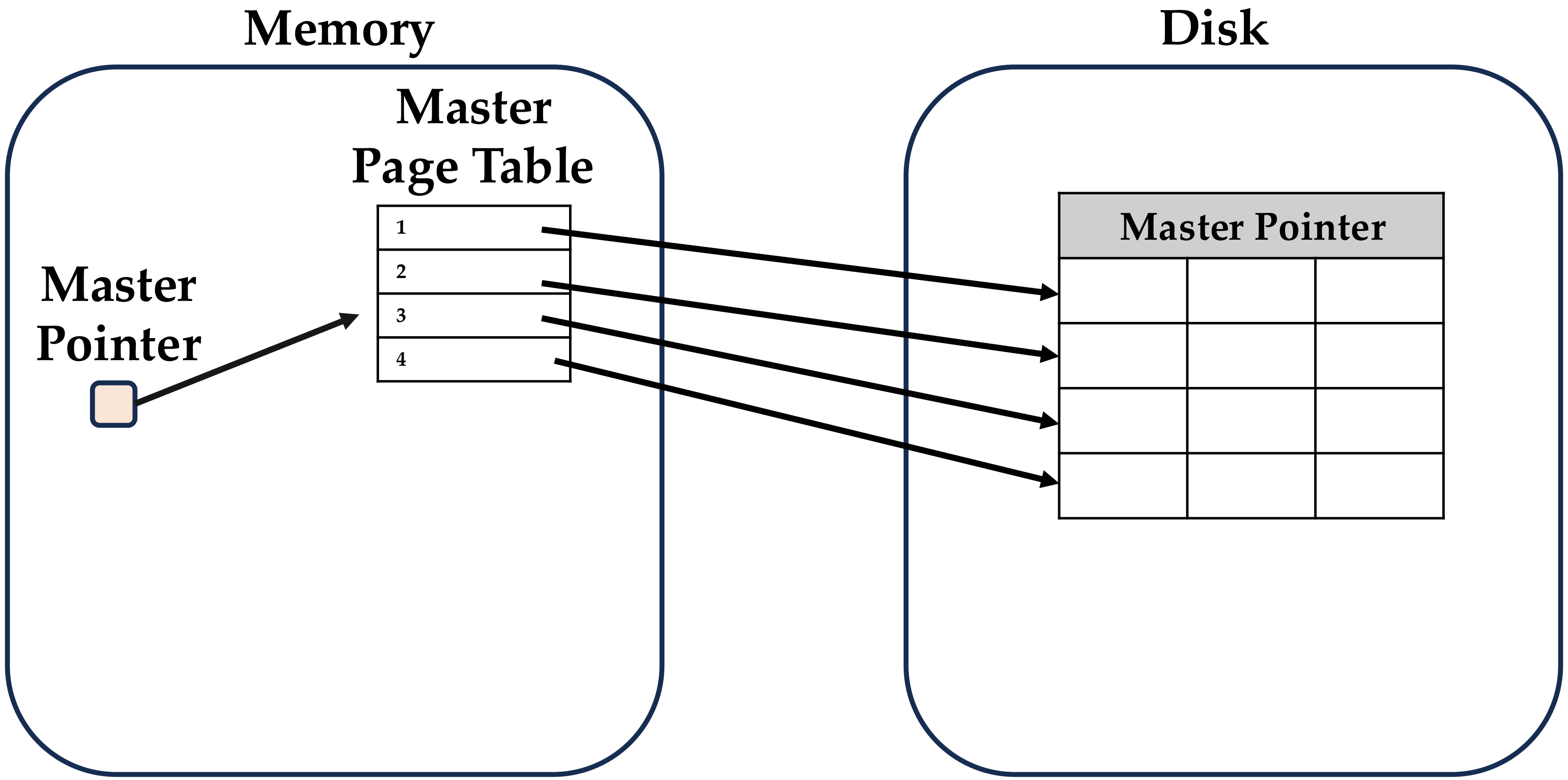
- DBMS maintains copies of pages.
    - Specifically, at most two versions of a page:

- **Master version** → Contains only changes from committed transactions.

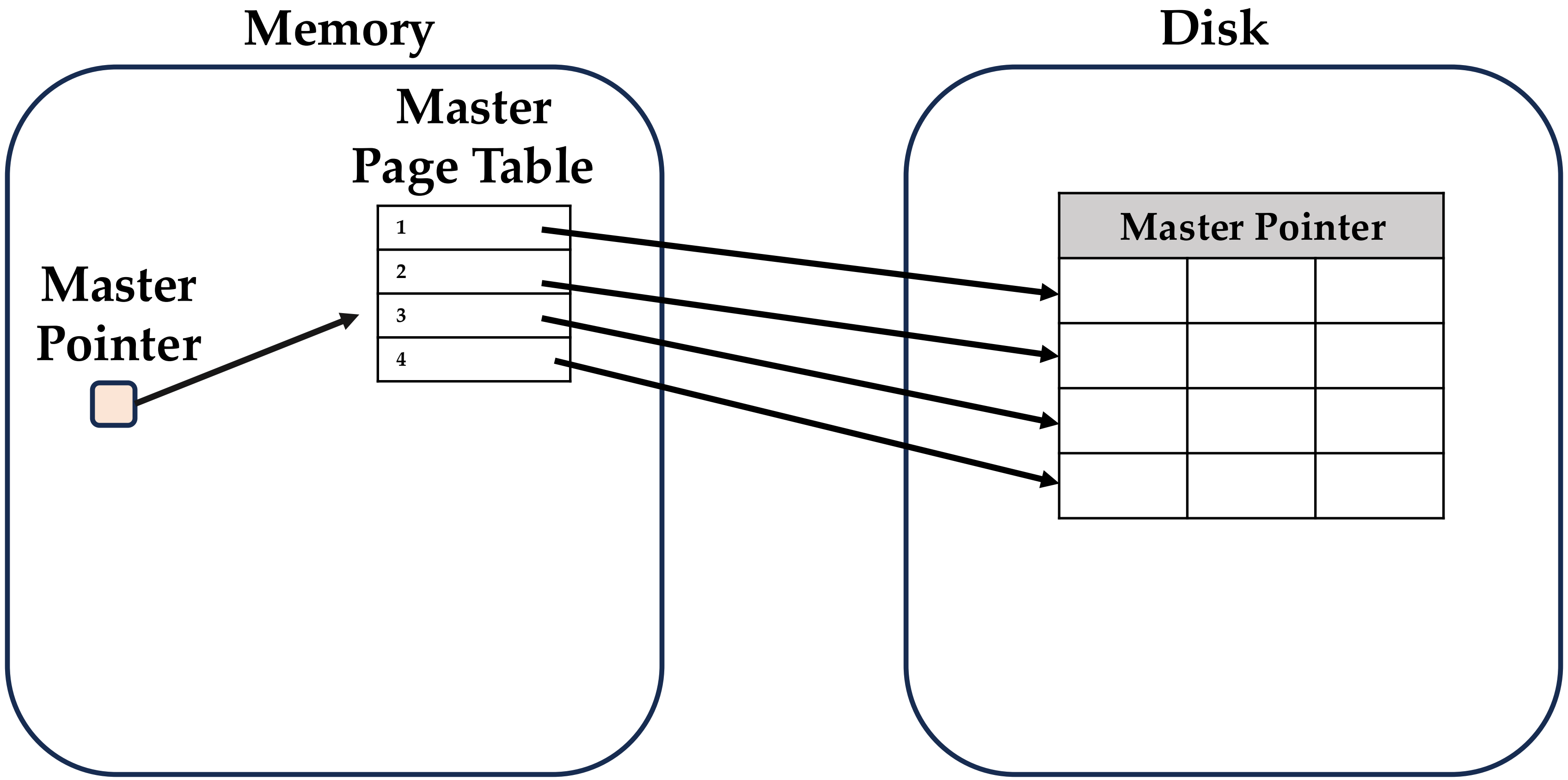
- **Shadow version** → Updates made by uncommitted transactions.
    - If a transaction wants to write/update, create a shadow version.

- To install updates when a transaction commits, overwrite the root so it points to the shadow.
    - **Swapping master and shadow versions**.

Shadow Paging

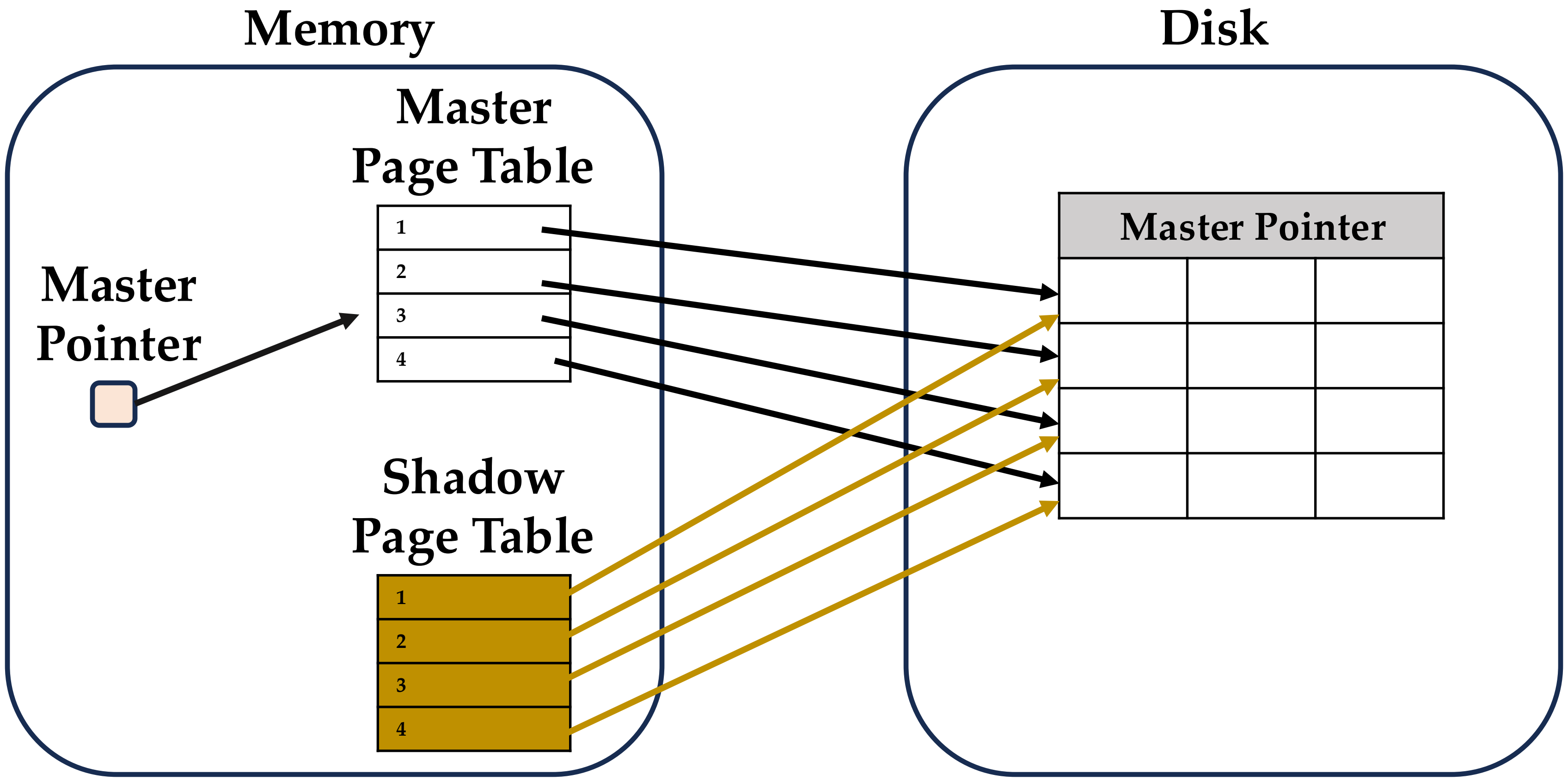


Shadow Paging



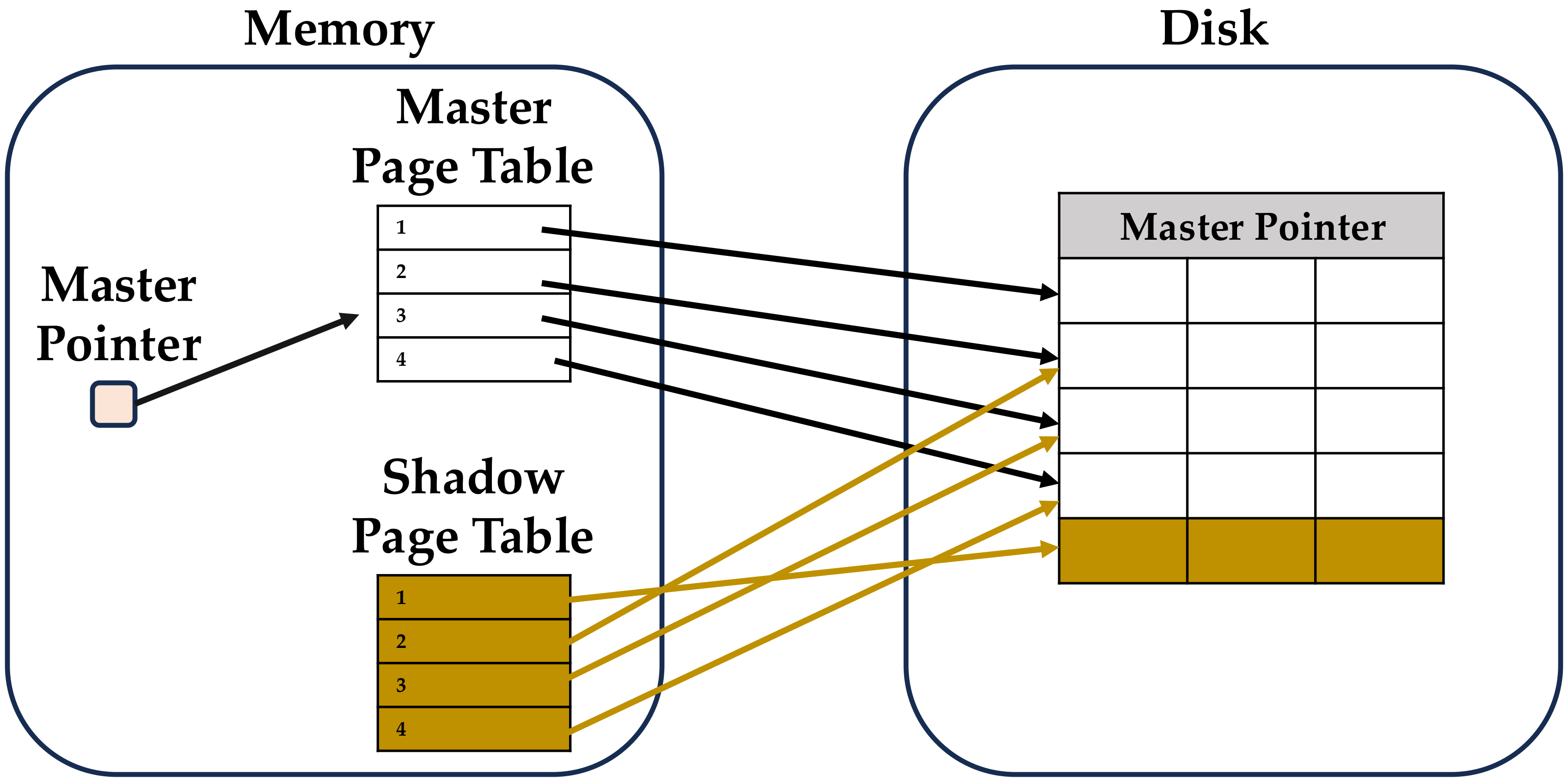
Say a transaction T1 wants to update some of these pages.

Shadow Paging



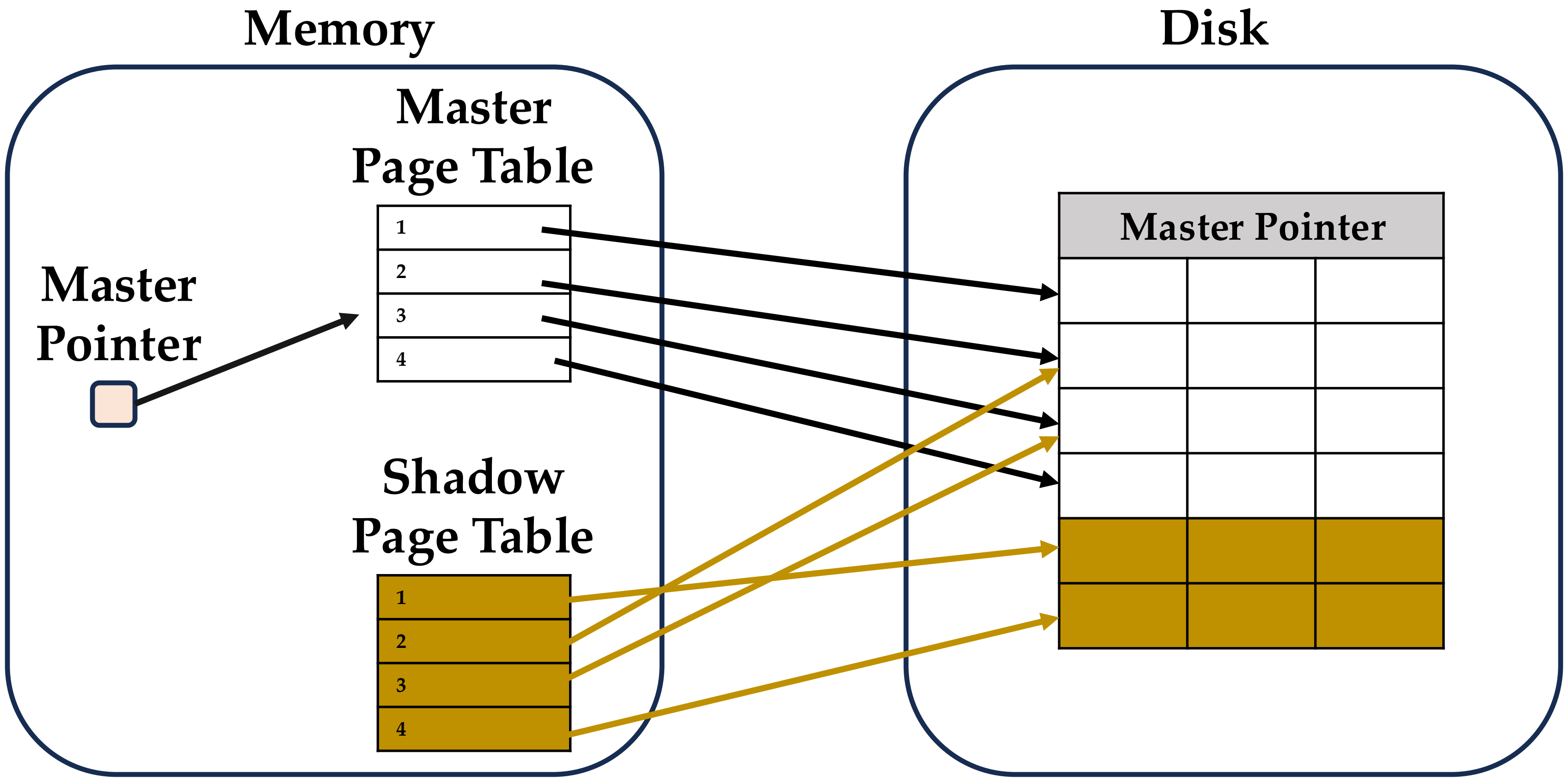
So, create a shadow page table that points to original pages in the disk.

Shadow Paging



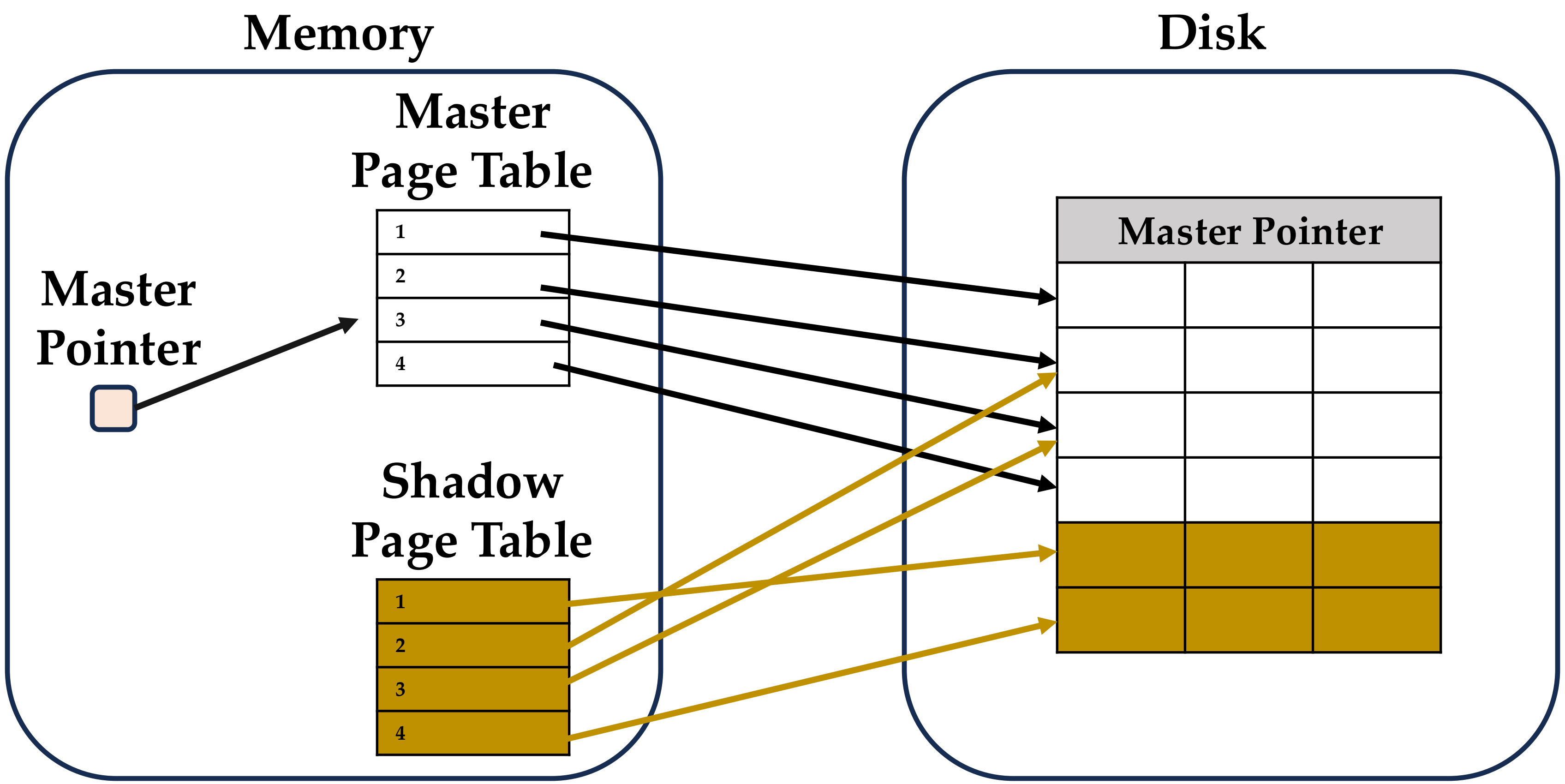
Now, say T1 wants to write a record in Page 1, so create a new Page in disk

Shadow Paging



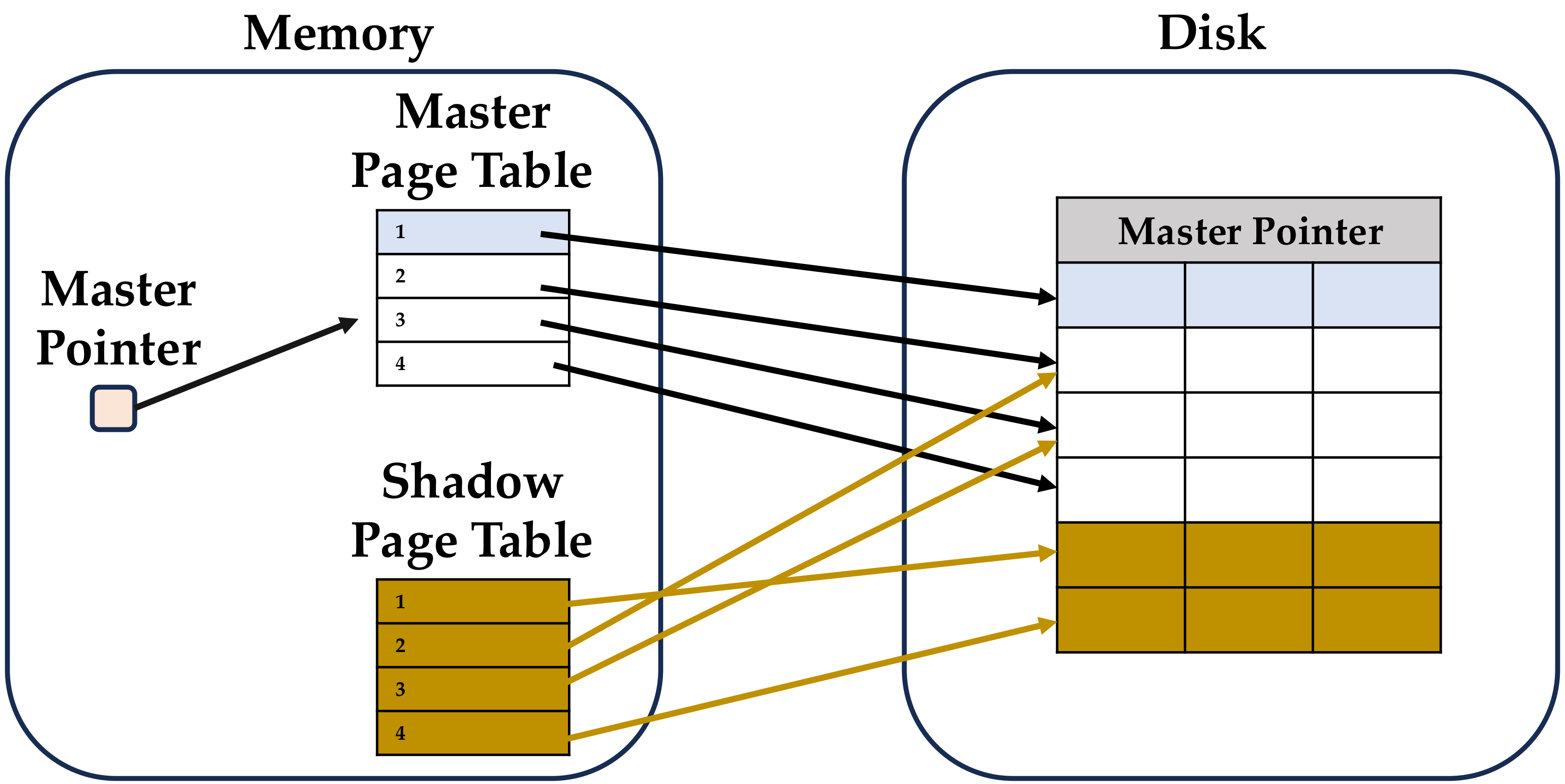
Now, say T1 wants to write a record in Page 4, so create a new Page in disk

Shadow Paging



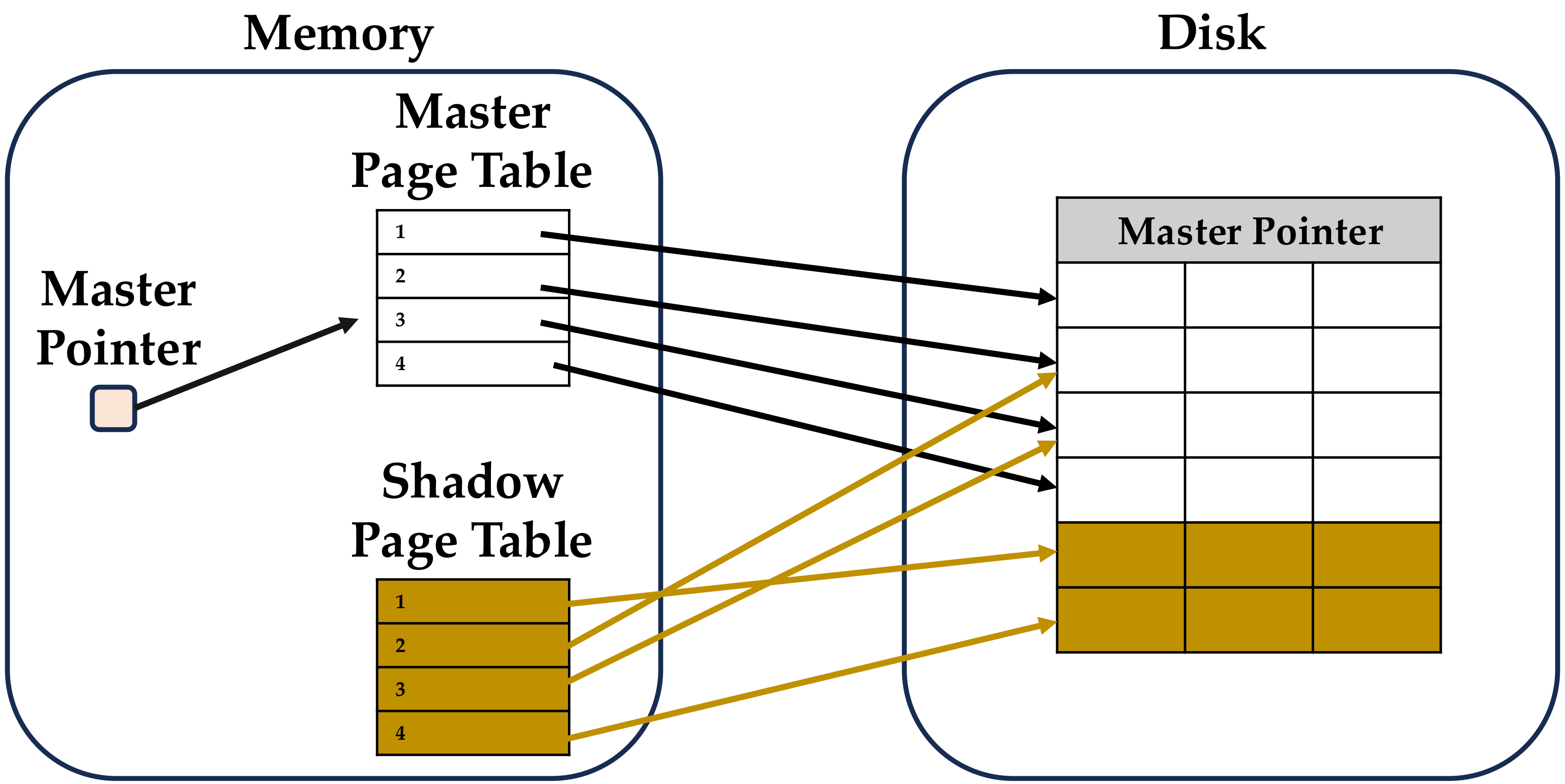
Now, say a transaction T2 wants to read a record from Page 1, where should it read from?

Shadow Paging



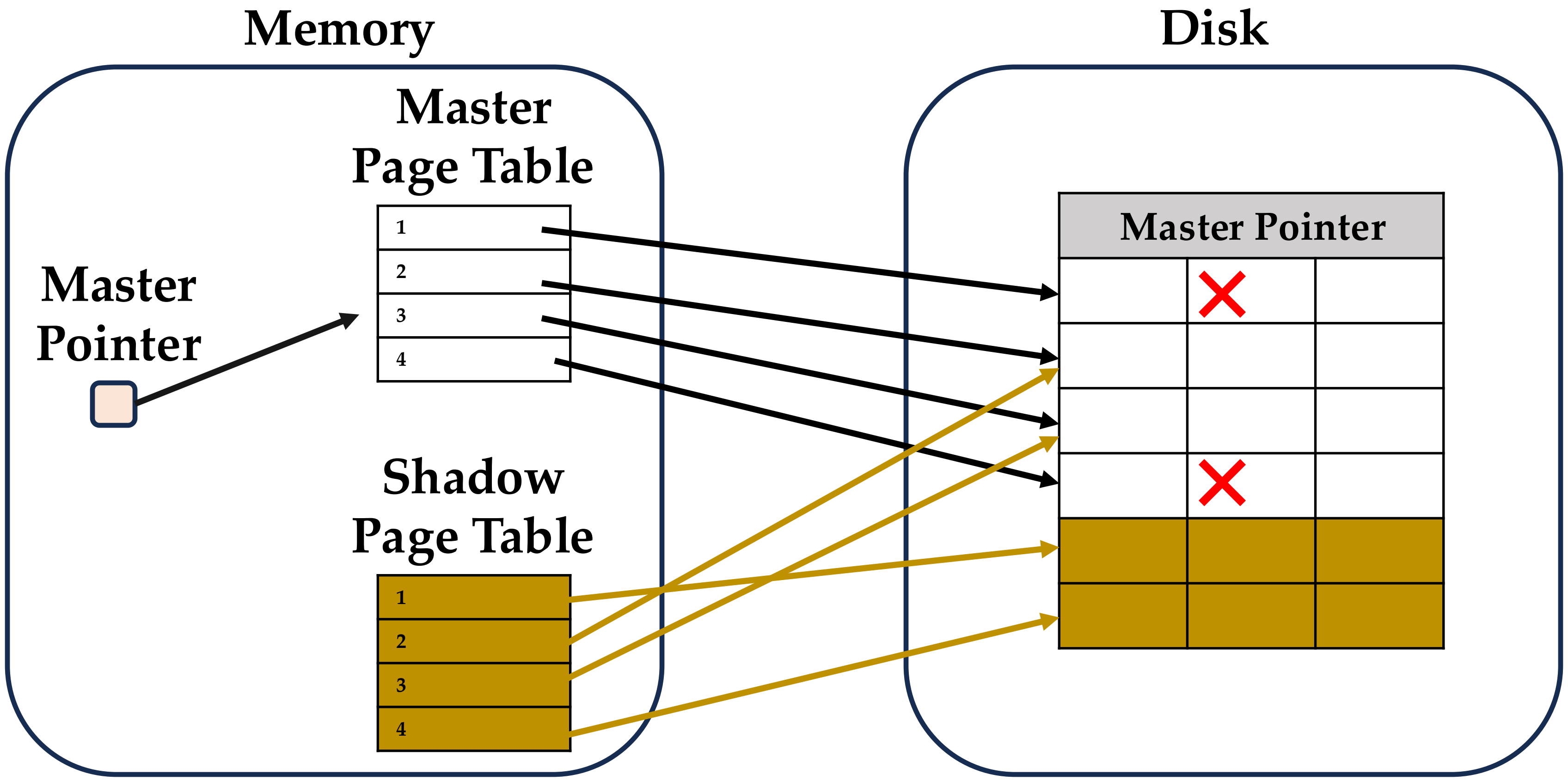
Transaction T2 will follow the Master Pointer and read Page 1 from disk

Shadow Paging



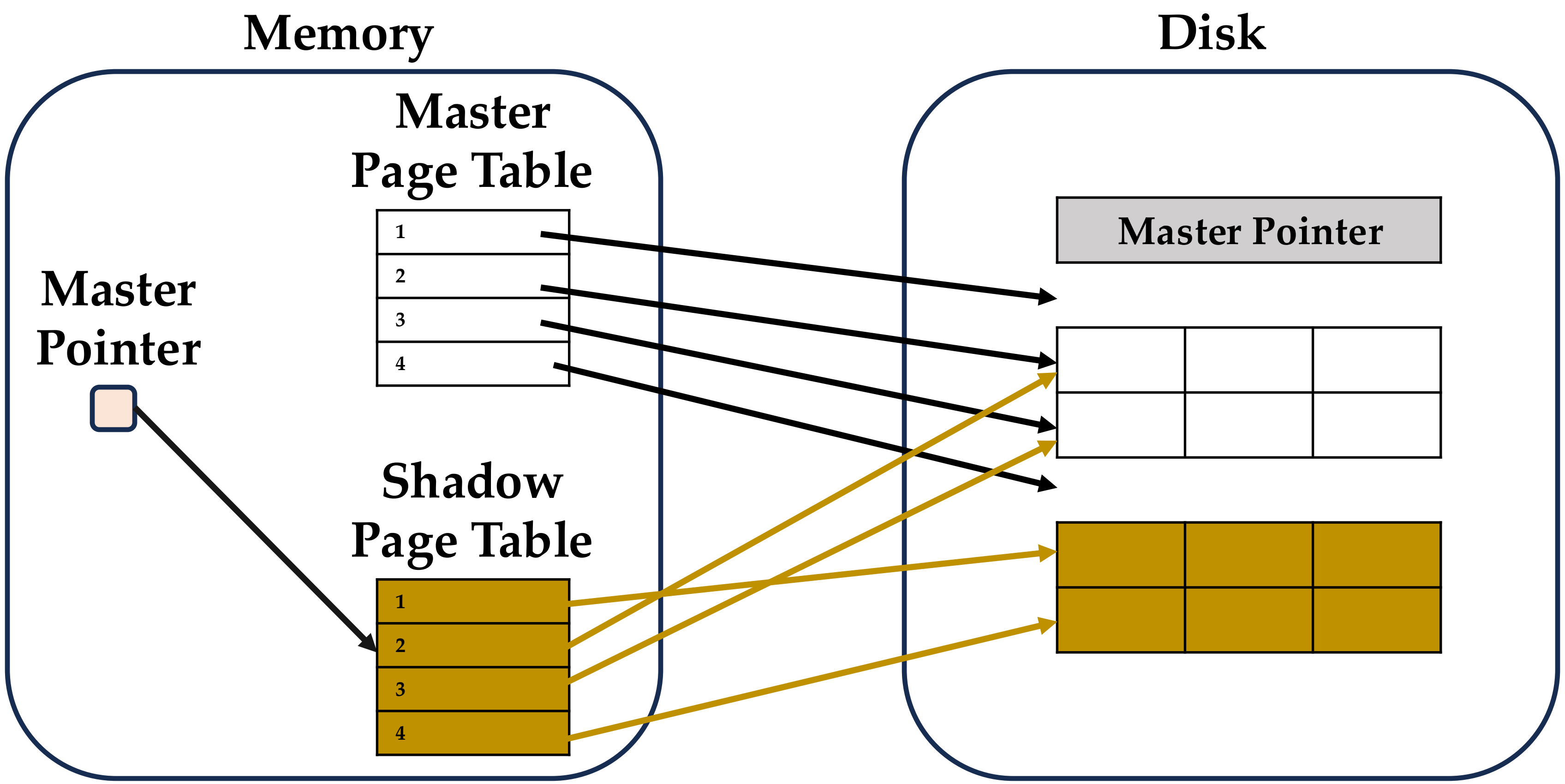
Now, what happens T1 commits?

Shadow Paging



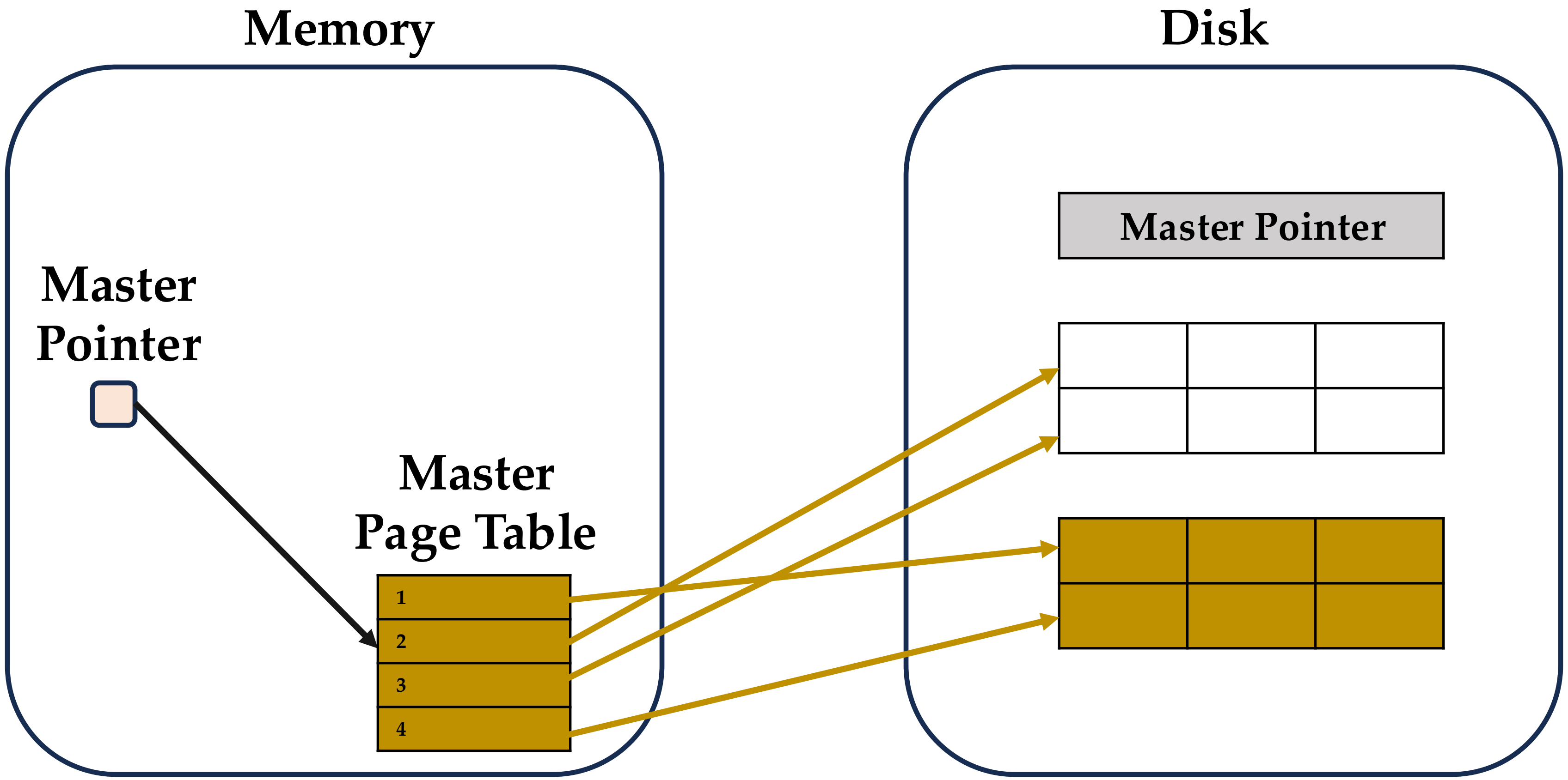
Original Pages for 1 and 4 are deleted.

Shadow Paging



And, the master pointer points to the shadow page table.

Shadow Paging



And, the shadow page table becomes the master page table.

Shadow Paging

- Supporting rollbacks and recovery is easy with shadow paging.

- **Undo:**
  - Remove the shadow pages.
  - Leave the master and the DB root pointer alone.

- **Redo:**
  - Not needed at all.

Shadow Paging Disadvantages

- Copying the entire page table is expensive.
- Commit overhead is high.

Steal + No-Force

- Allow transactions to read uncommitted updates and do not force writing updates to the disk.

- **How to design a steal + no-force policy**?

Write-Ahead Log (WAL)

- Maintain a **log file** separate from buffer pool that tracks the changes that transactions make to that database.
- **Assumption →** the log file is on stable storage.
- Log contains enough information to perform the necessary undo and redo operations on the database.
- **DBMS must write to disk the log file before it can flush the actual object to disk.**
- A transaction is not considered committed until all its log records have been written to stable storage!

WAL Protocol

- Write a **<Begin>** record to the log for each transaction to mark its starting point.
- **Append a record** every time a transaction changes an object:
  - Transaction Id
  - Object Id
  - Before Value (Undo)
  - After Value (Redo)
- When a transaction finishes, the DBMS appends a **<Commit>** record to the log.
- Make sure that all log records are flushed to disk before marking a transaction committed.

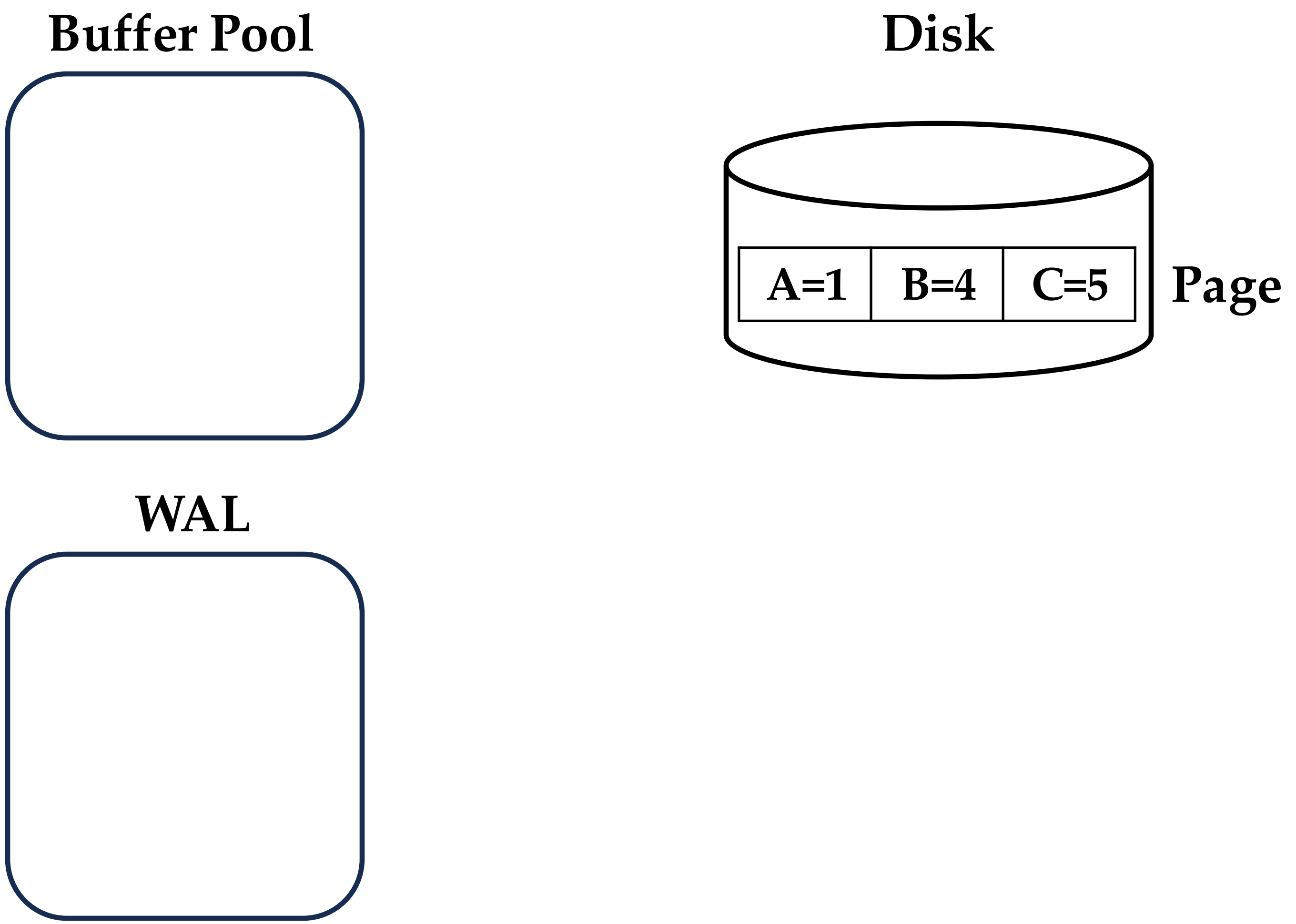
WAL Protocol

Time

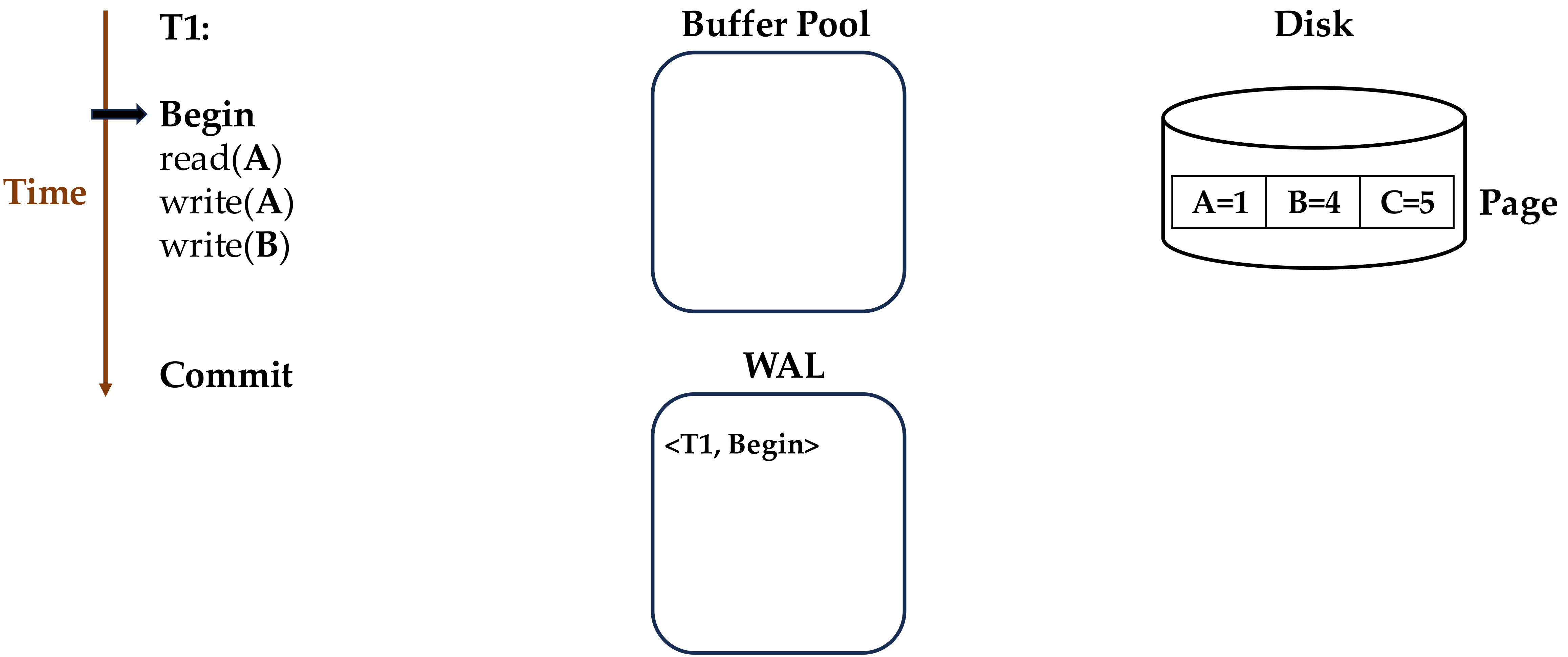
T1:

Begin
read(**A**)
write(**A**)
write(**B**)

Commit

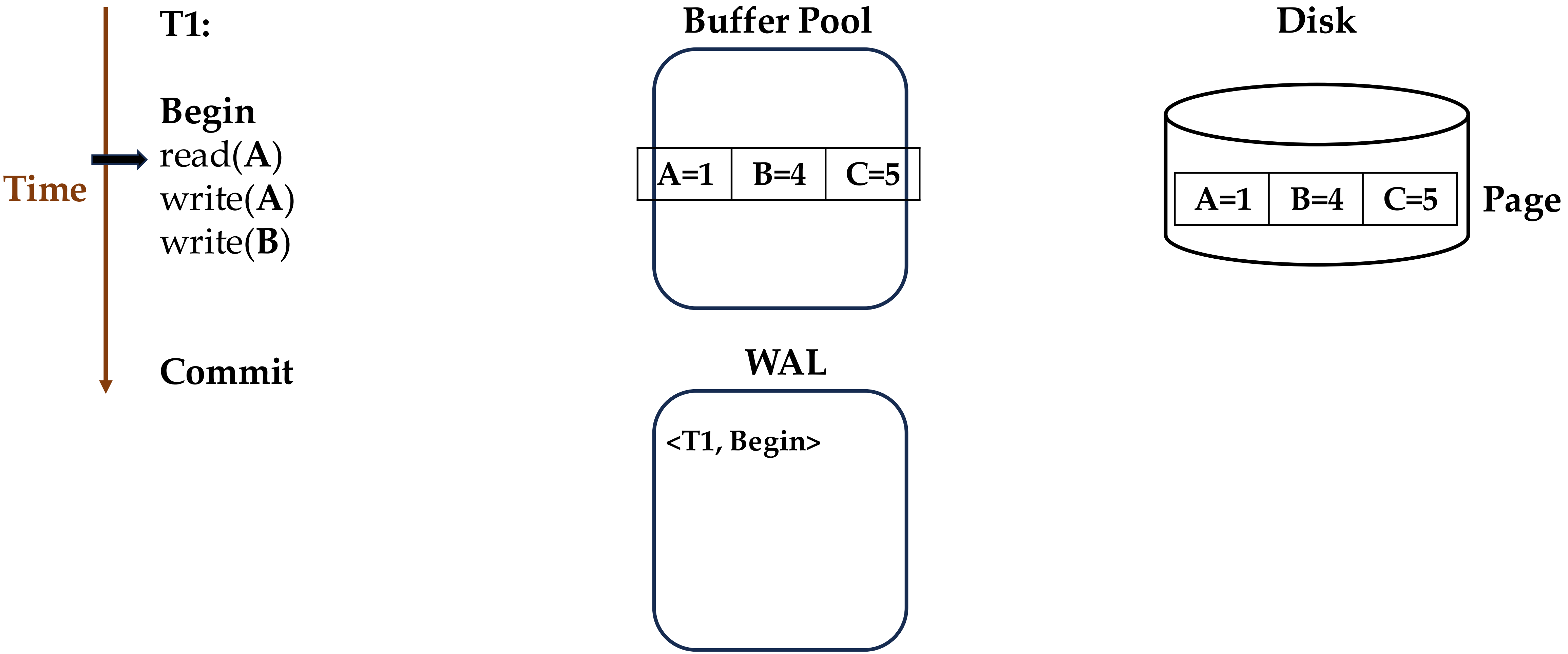


WAL Protocol



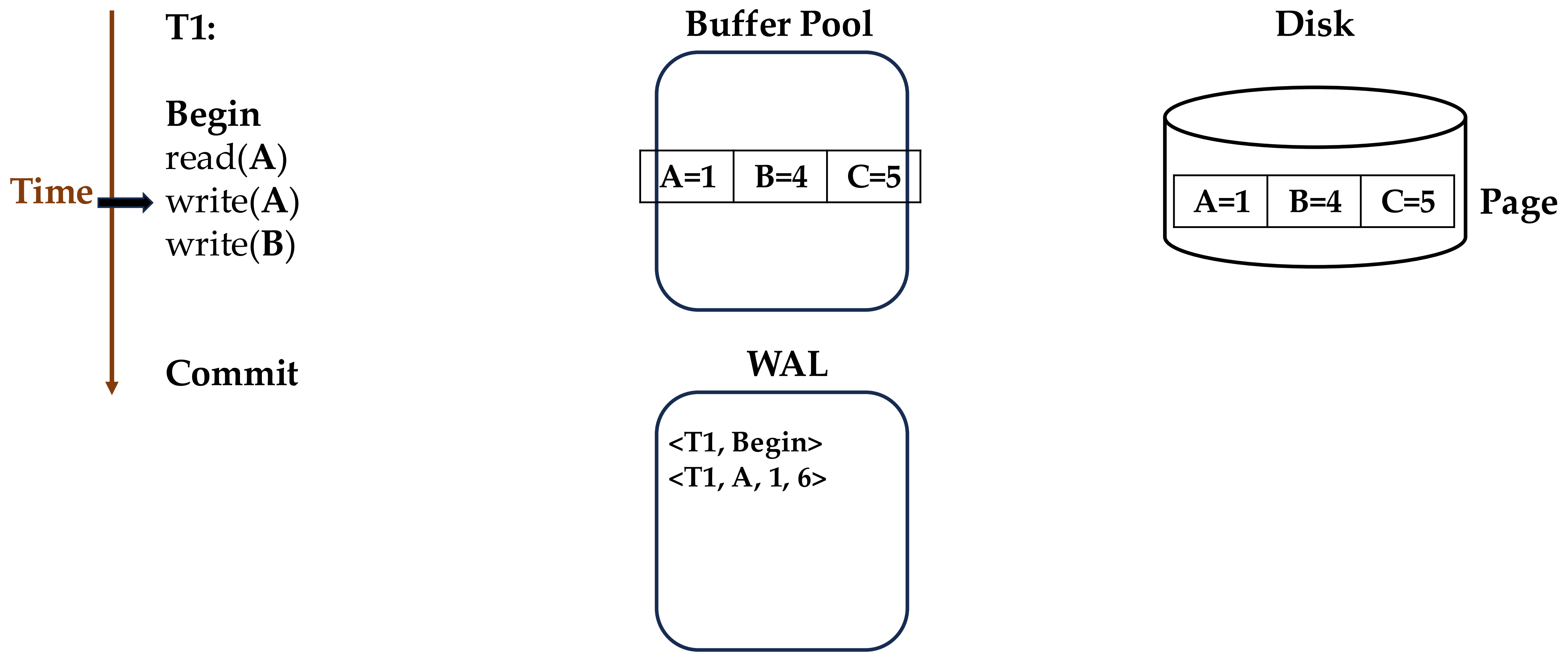
Add a transaction start entry to WAL.

WAL Protocol



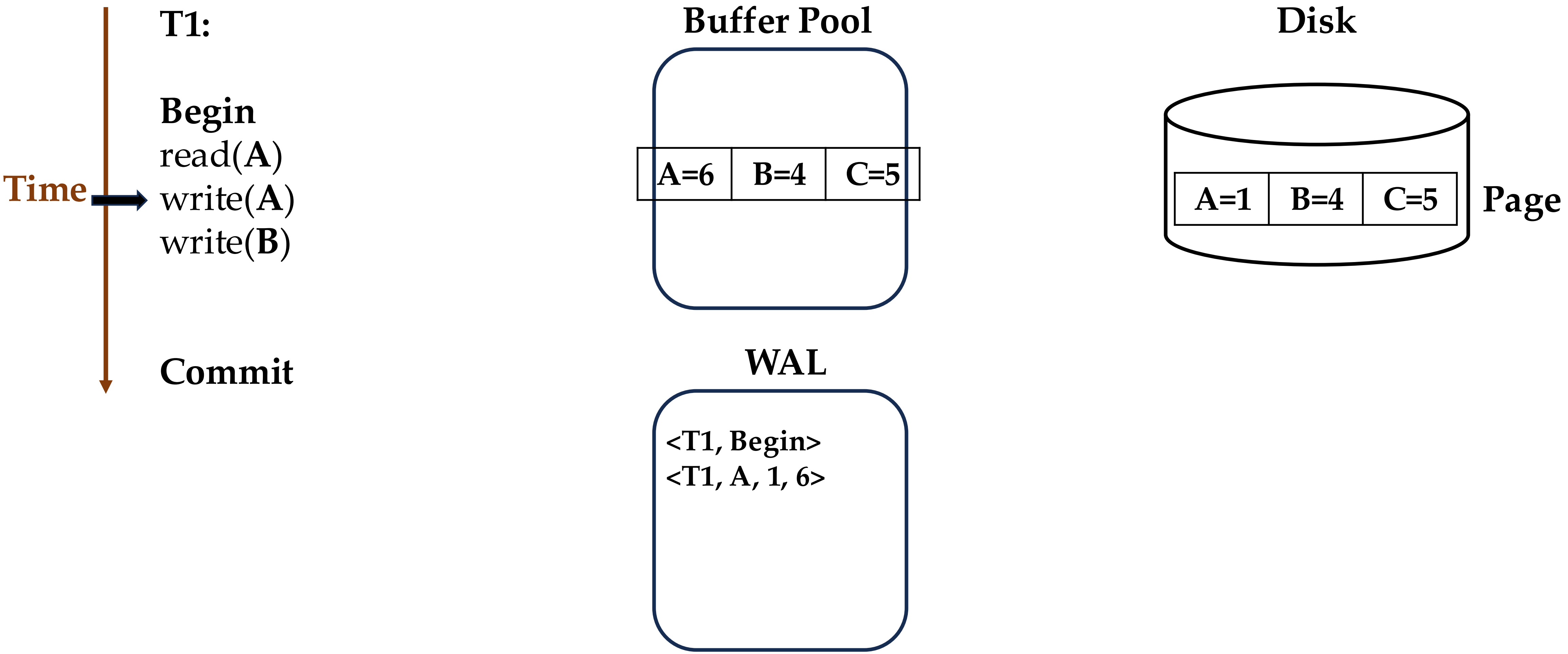
Read the page to the buffer.

WAL Protocol



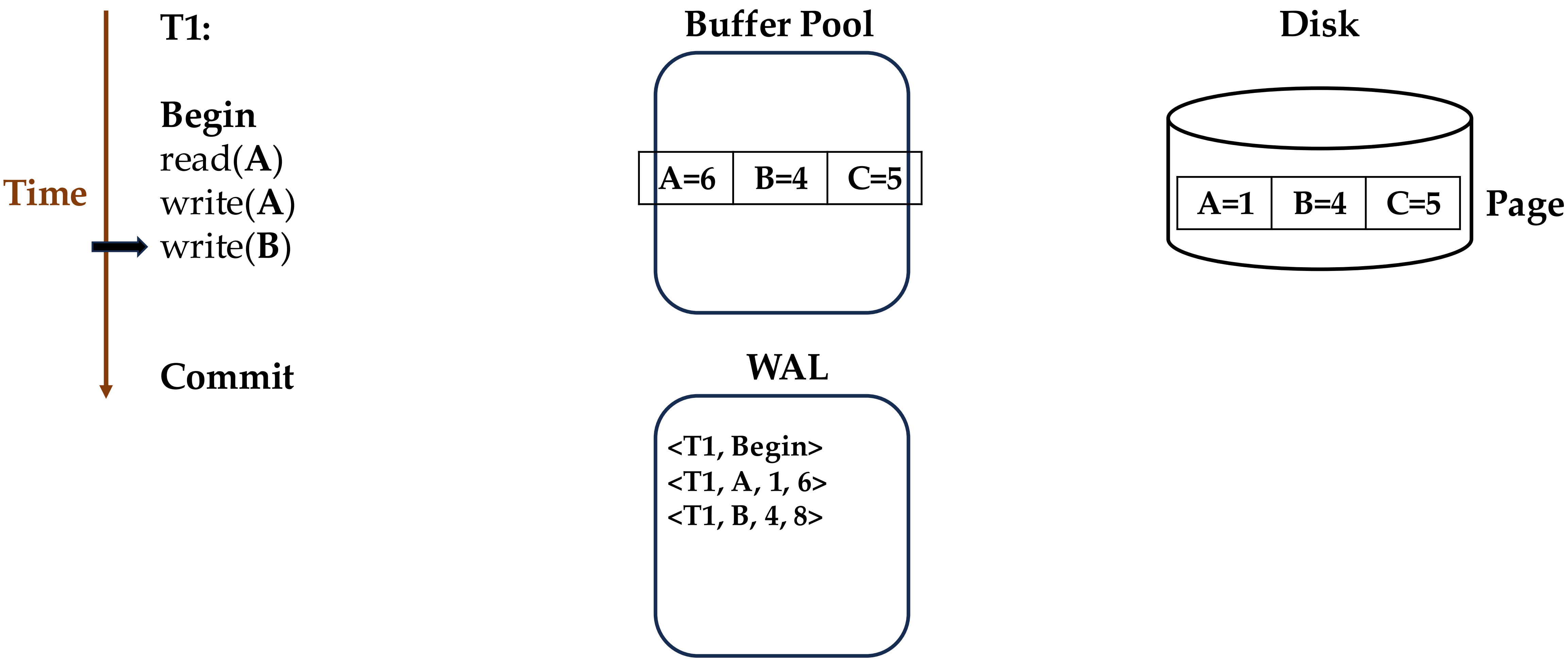
Say, this write to A updates its value from 1 to 6 → First, add an entry to WAL.

WAL Protocol



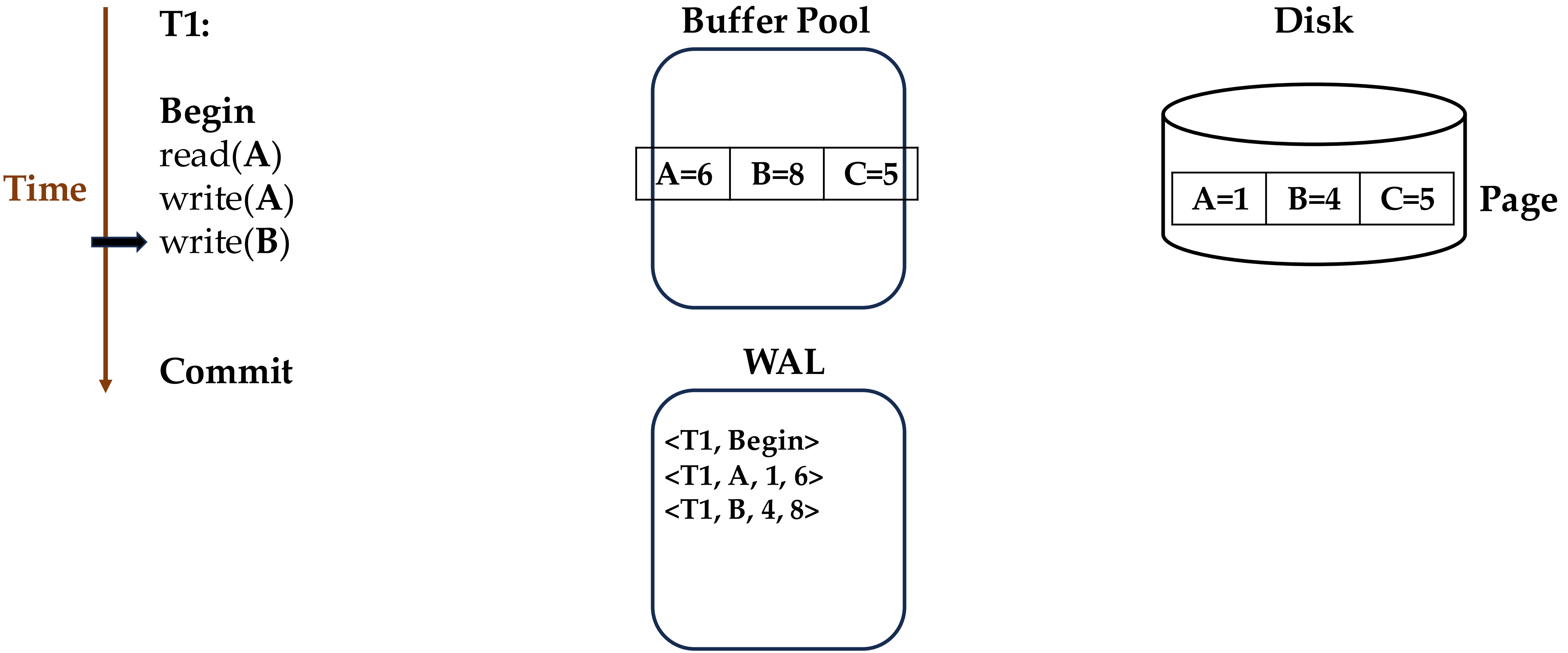
Then update Buffer Pool.

WAL Protocol



Similarly, for B

WAL Protocol



Similarly, for B

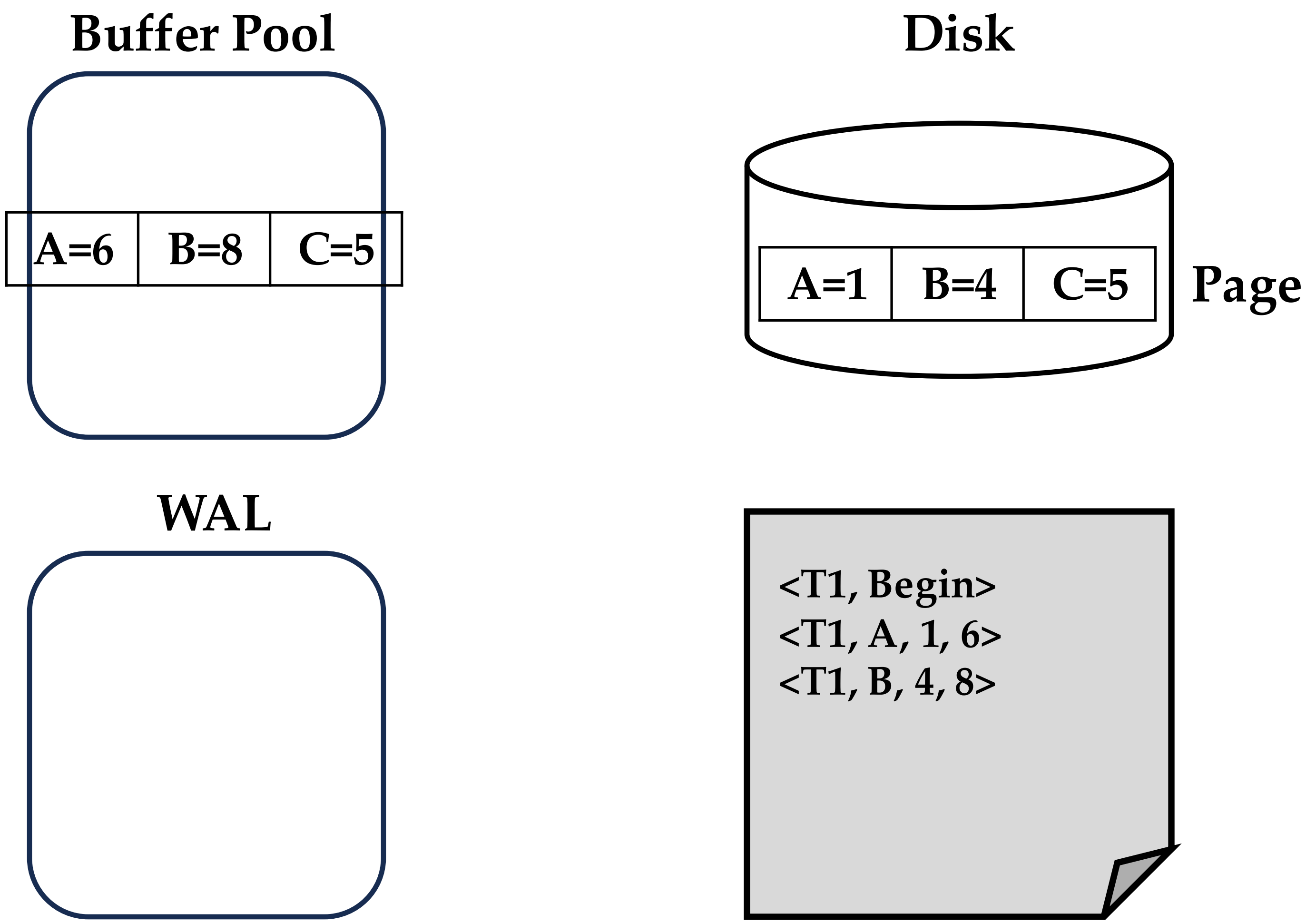
WAL Protocol

Time

T1:

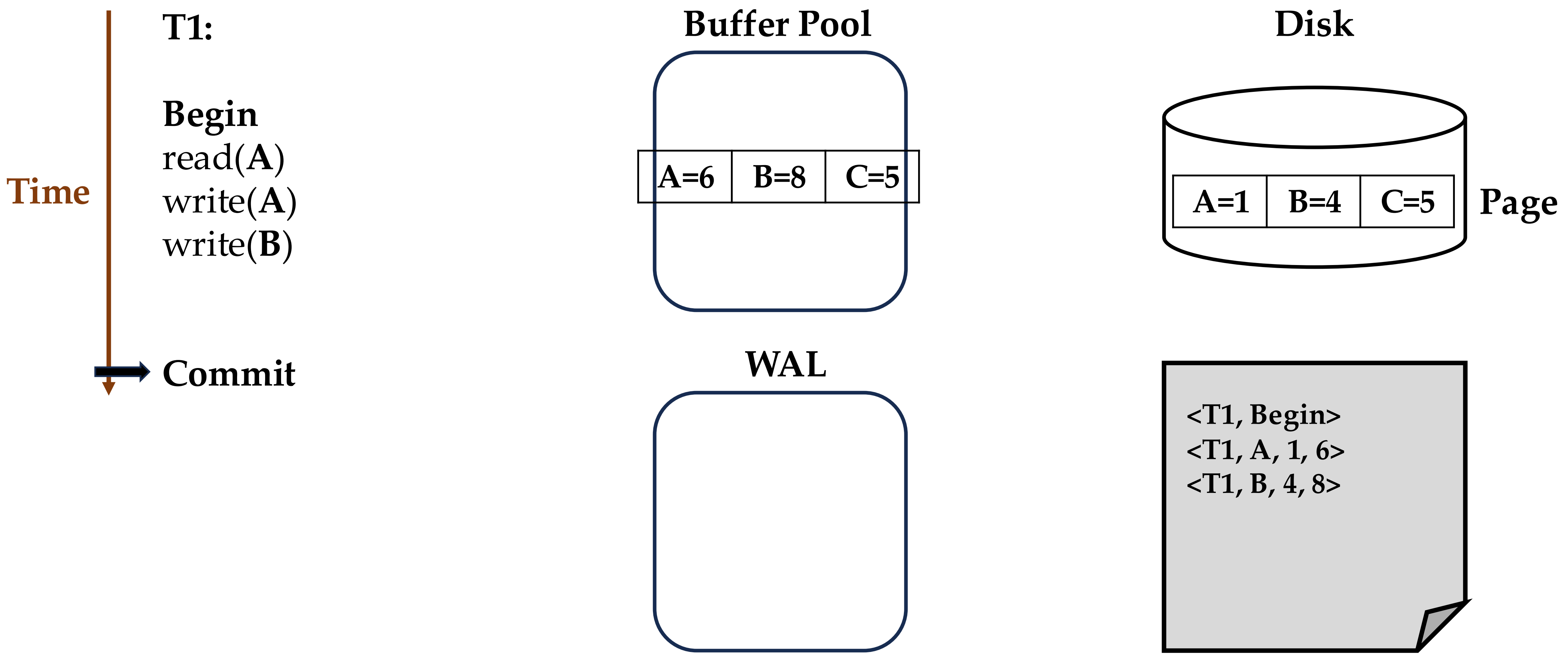
Begin
read(**A**)
write(**A**)
write(**B**)

Commit



On commit, first write WAL entries to the disk.

WAL Protocol



At this point, before even writing buffer pool to disk, any other transaction can access.

WAL Disadvantages

- What are the disadvantages for WAL?

WAL Disadvantages

- What are the disadvantages for WAL?

- Flushing the log buffer to disk every time a transaction commits bottlenecks the system.

- Solution?

WAL Disadvantages

- What are the disadvantages for WAL?

- Flushing the log buffer to disk every time a transaction commits bottlenecks the system.

- Solution?

- **Group Commits!**

WAL Disadvantages

- What are the disadvantages for WAL?

- Flushing the log buffer to disk every time a transaction commits bottlenecks the system.

- Solution?

- **Group Commits!**

WAL Group Commit

- Group Commits allows the DBMS to **amortize the overhead**.
- **Batch multiple log flushes together**.
- When the buffer is full → flush it to disk.
- Alternatively, use a timeout period (e.g., 1 second).

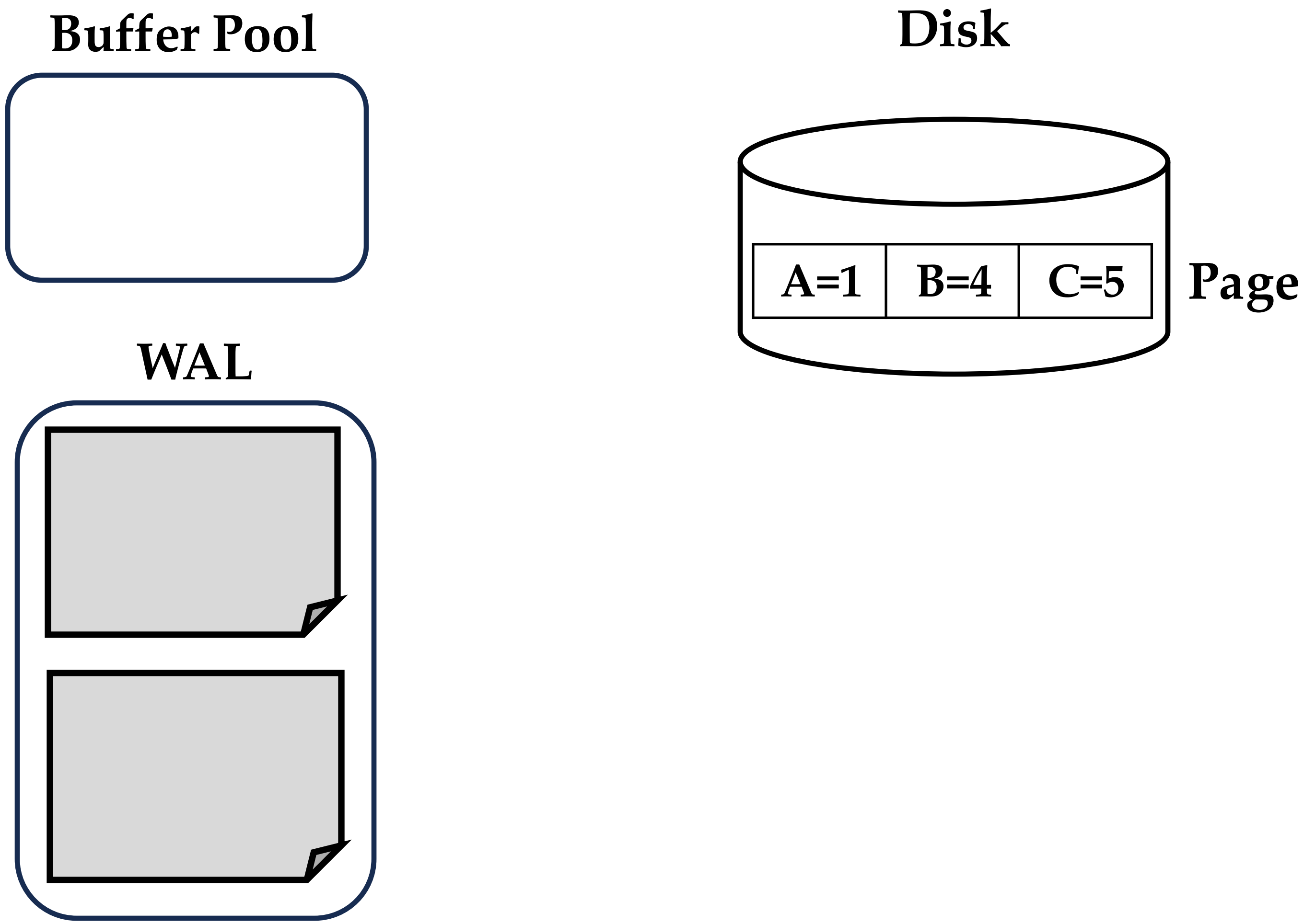
WAL Group Commit

- Group Commits allows the DBMS to **amortize the overhead**.
- **Batch multiple log flushes together**.
- When the buffer is full → flush it to disk.
- Alternatively, use a timeout period (e.g., 1 second).

WAL Group Commit

Time

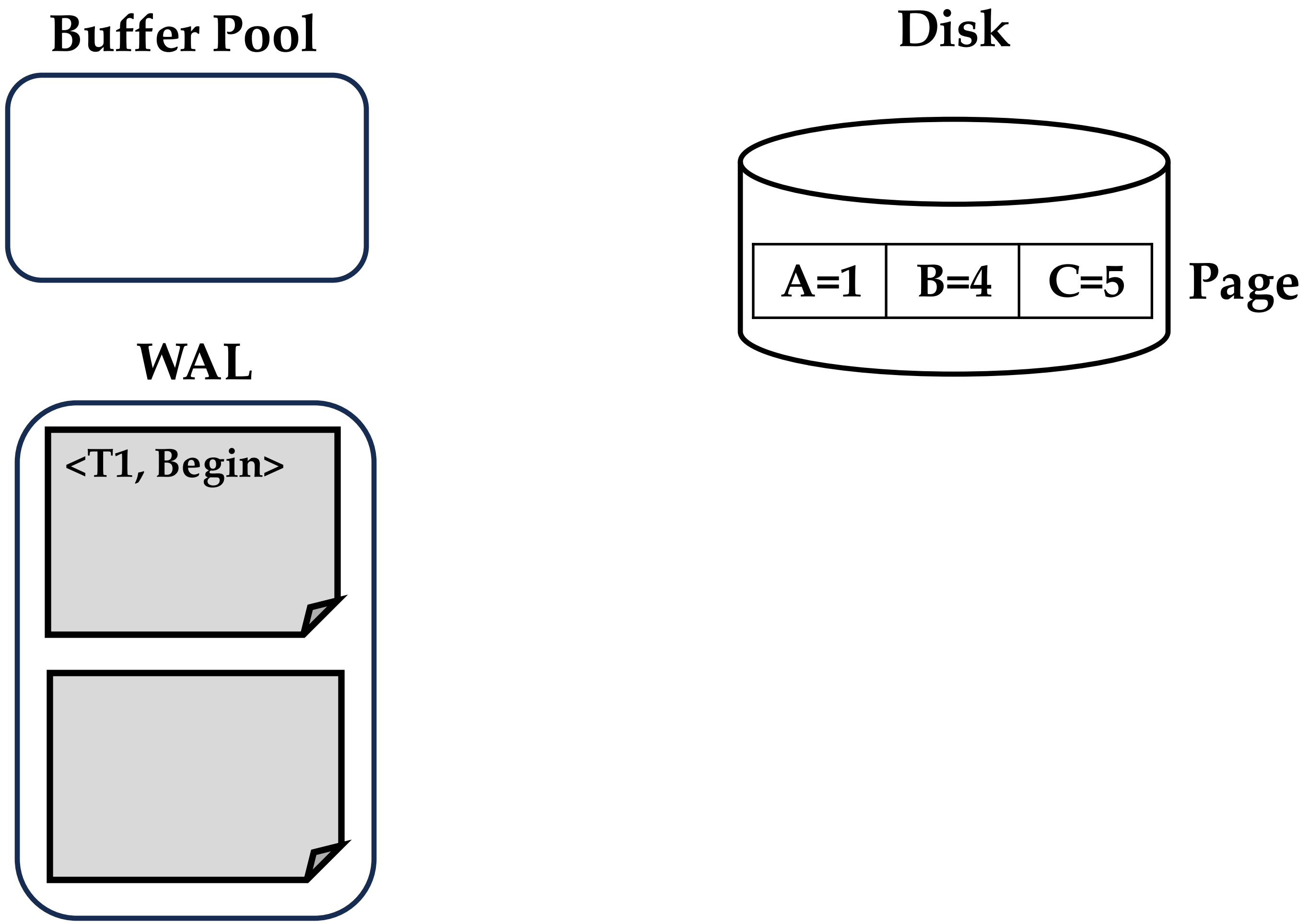
T1:	T2
Begin	
read(**A**)	
write(**A**)	
write(**B**)	
	Begin
	read**(C)**
	write**(C)**
Commit	
	Commit



WAL Group Commit

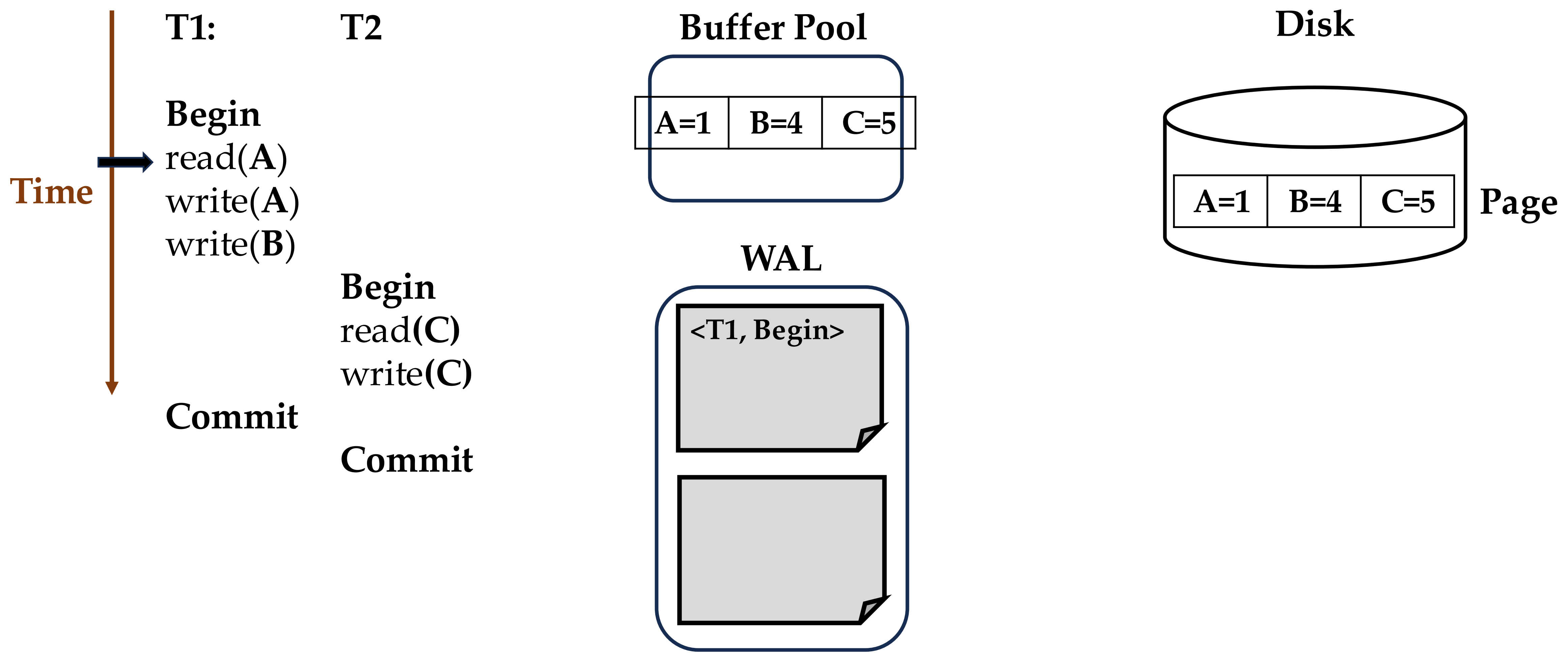
T1:	T2
Begin	
read(**A**)	
write(**A**)	
write(**B**)	
	Begin
	read**(C)**
	write**(C)**
Commit	
	Commit

Time

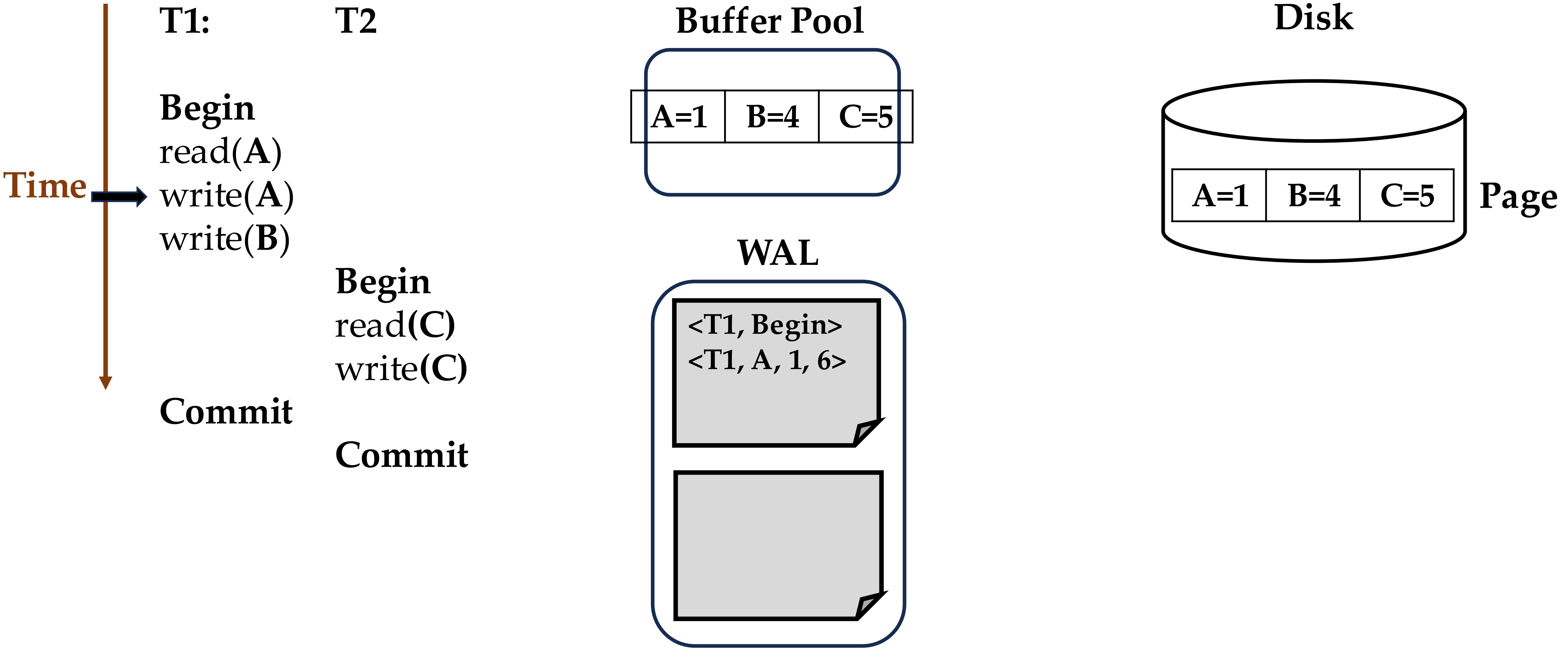


Add a transaction start entry to WAL.

WAL Group Commit



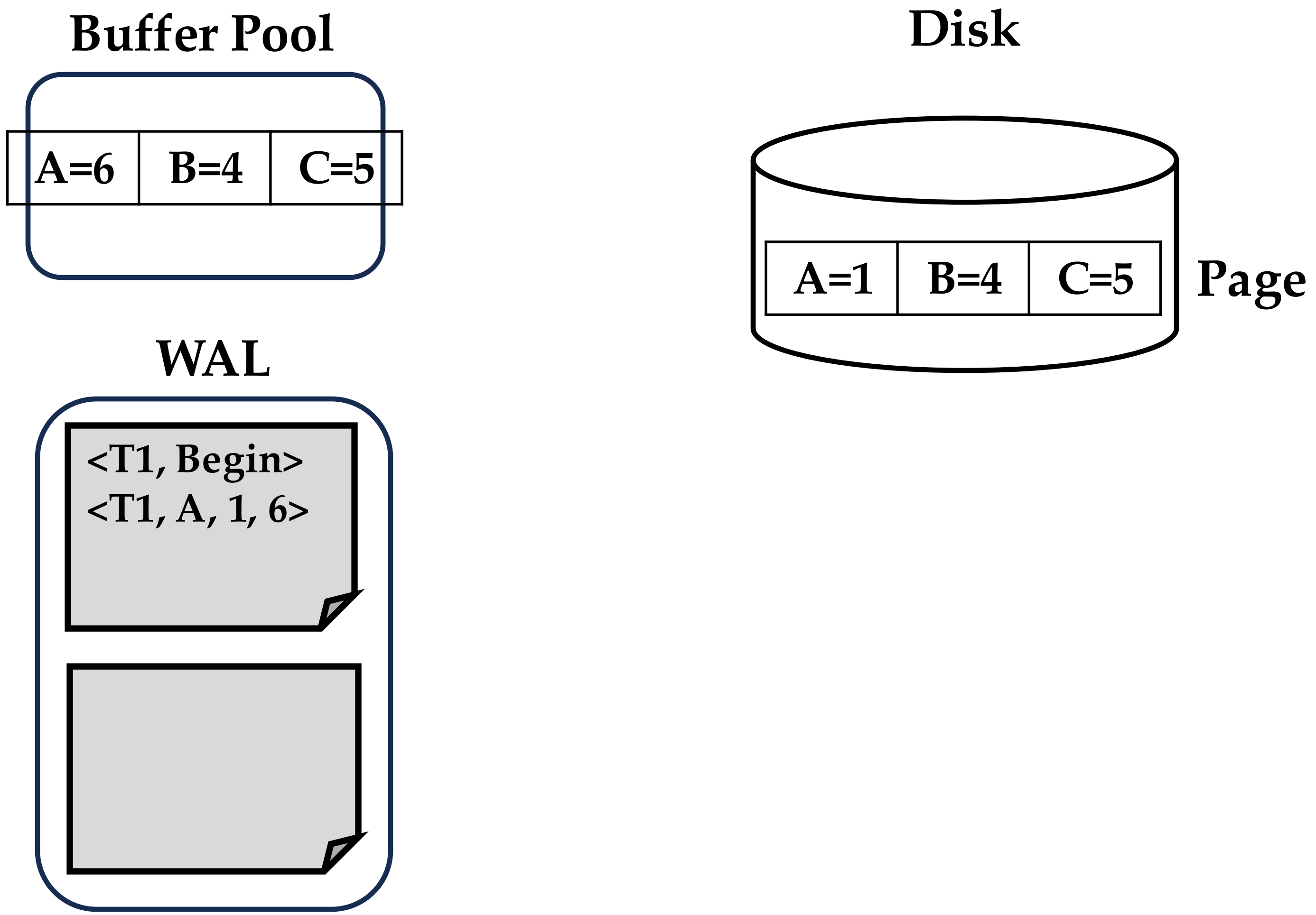
WAL Group Commit



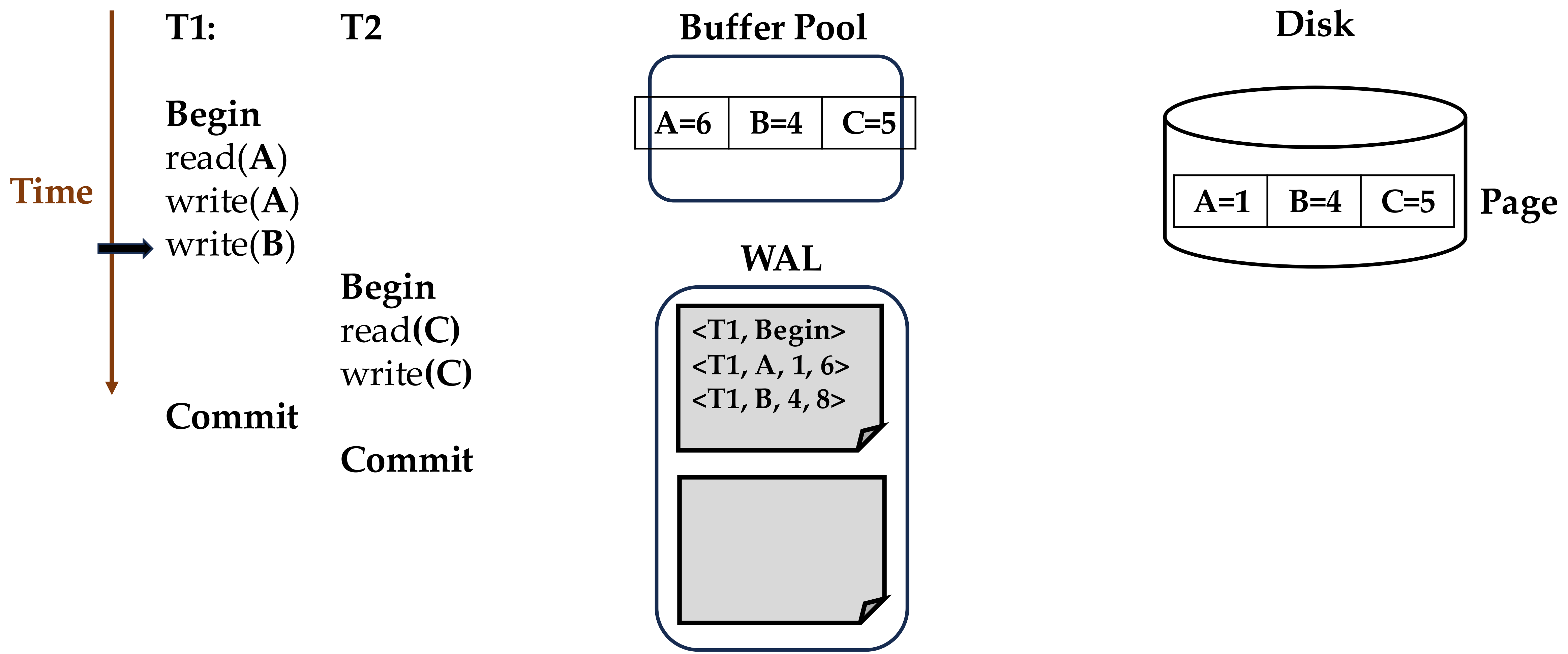
WAL Group Commit

Time

T1:	T2
Begin	
read(**A**)	
write(**A**)	
write(**B**)	
	Begin
	read**(C)**
	write**(C)**
Commit	
	Commit



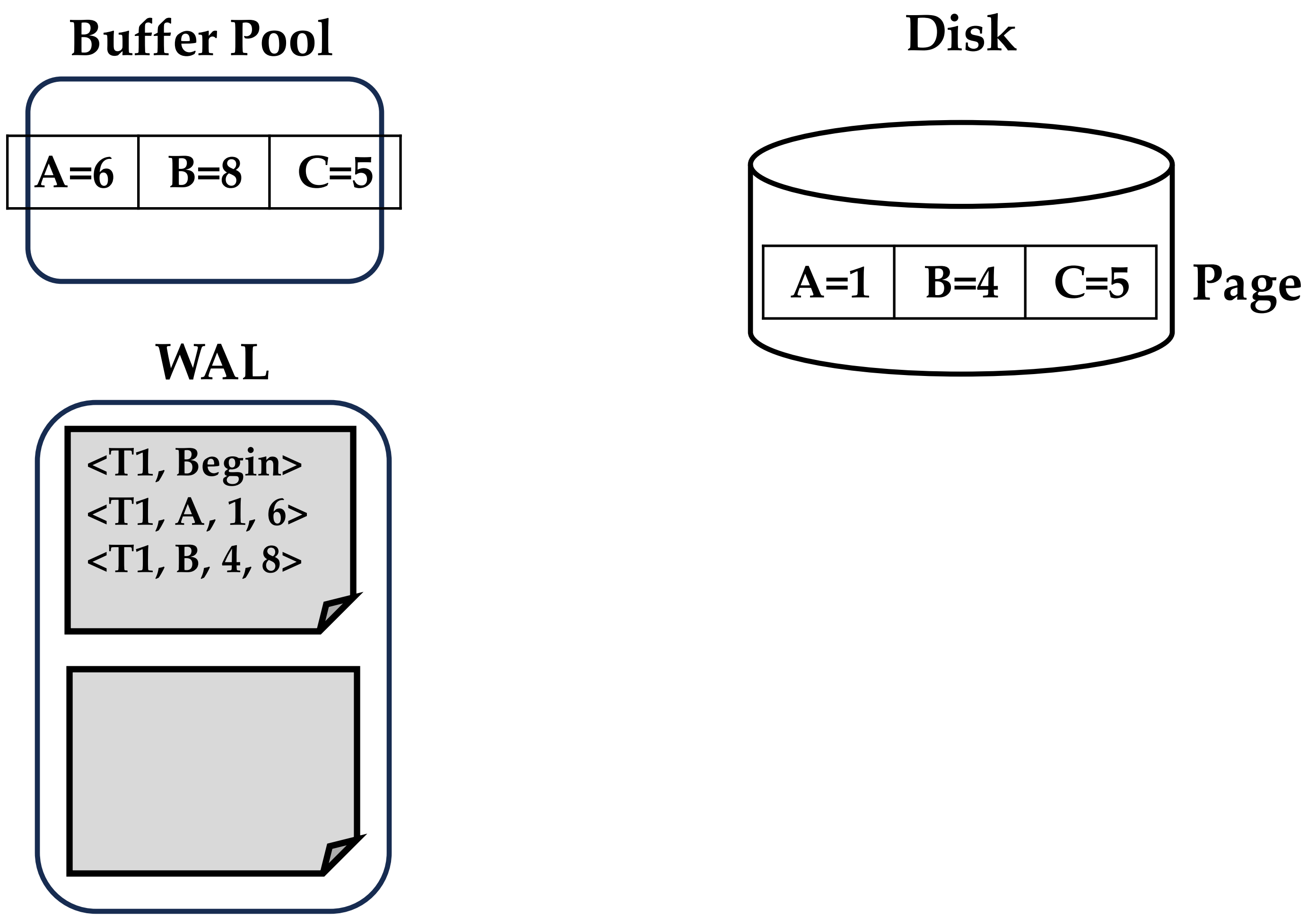
WAL Group Commit



WAL Group Commit

Time

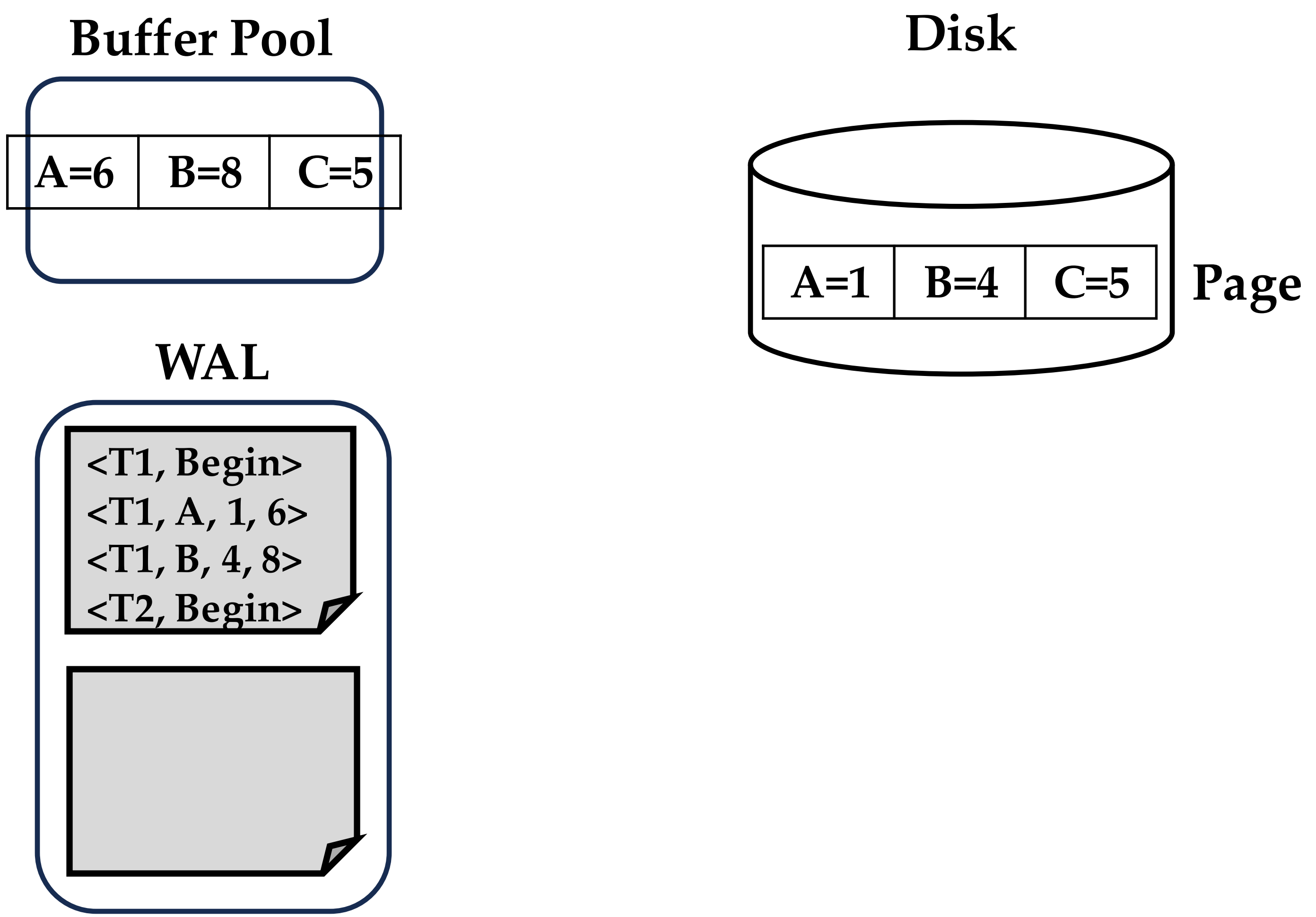
T1:	T2
Begin	
read(**A**)	
write(**A**)	
write(**B**)	
	Begin
	read(**C**)
	write(**C**)
Commit	
	Commit



WAL Group Commit

Time

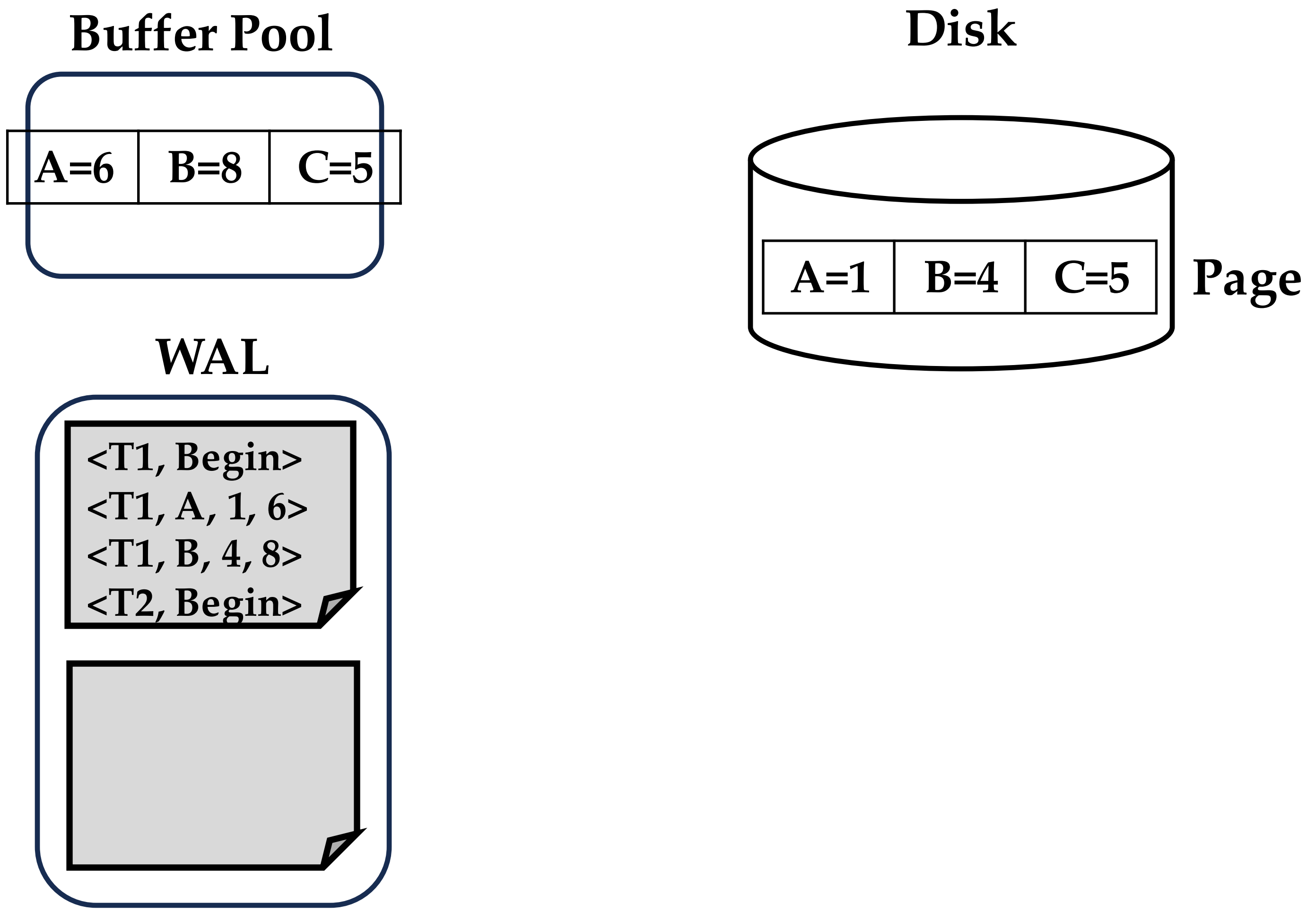
T1:	T2
Begin	
read(**A**)	
write(**A**)	
write(**B**)	
	Begin
	read**(C)**
	write**(C)**
Commit	
	Commit



WAL Group Commit

Time

T1:	T2
Begin	
read(A)	
write(A)	
write(B)	
	Begin
	read(C)
	write(C)
Commit	
	Commit



WAL Group Commit

Time

T1:	T2
Begin	
read(**A**)	
write(**A**)	
write(**B**)	
	Begin
	read(**C**)
	write(**C**)
Commit	
	Commit